Brain-on-Switch: Towards Advanced Intelligent Network Data Plane via NN-Driven Traffic Analysis at Line-Speed

Jinzhu Yan[1,*] Haotian Xu[1,*] Zhuotao Liu[1,2,✉] Qi Li[1,2] Ke Xu[1,2]
Mingwei Xu[1,2] Jianping Wu[1,2]
[1] *Tsinghua University* [2] *Zhongguancun Laboratory*

Abstract

The emerging programmable networks sparked significant research on Intelligent Network Data Plane (INDP), which achieves learning-based traffic analysis at line-speed. Prior art in INDP focus on deploying tree/forest models on the data plane. We observe a fundamental limitation in tree-based INDP approaches: although it is possible to represent even larger tree/forest tables on the data plane, the flow features that are computable on the data plane are fundamentally limited by hardware constraints. In this paper, we present BoS to push the boundaries of INDP by enabling Neural Network (NN) driven traffic analysis at line-speed. Many types of NNs (such as Recurrent Neural Network (RNN), and transformers) that are designed to work with sequential data have advantages over tree-based models, because they can take raw network data as input without complex feature computations on the fly. However, the challenge is significant: the recurrent computation scheme used in RNN inference is fundamentally different from the match-action paradigm used on the network data plane. BoS addresses this challenge by (i) designing a novel data plane friendly RNN architecture that can execute unlimited RNN time steps with limited data plane stages, effectively achieving line-speed RNN inference; and (ii) complementing the on-switch RNN model with an off-switch transformer-based traffic analysis module to further boost the overall performance. We implement a prototype of BoS using a P4 programmable switch as our data plane, and extensively evaluate it over multiple traffic analysis tasks. The results show that BoS outperforms state-of-the-art in both analysis accuracy and scalability.

1 Introduction

The emerging programmable network hardware (*e.g.,* P4 switch [5], NetFPGA [36] and SmartNIC [13,37,54]) sparked significant research on Intelligent Network Data Plane (INDP). Compared with other *AI-assisted networking designs* which deploy learning models on either end-hosts (*e.g.,* congestion control [1,63]) or network control plane (including auxiliary servers) (*e.g.,* routing control [29,72]), INDP is *forwarding-native* since it deploys learning models directly on network data plane. Thus, the key merit of INDP, as first summarized in [71], is that it enables intelligent network traffic analysis at line-speed based on data-driven learning models rather than empirical rules/protocols.

The initial exploration of INDP begins with extracting fine-grained flow information from the programmable data plane to support a variety of overarching applications, such as covert channel detection [61], RTT measurement [45], traffic classification [4], and DDoS mitigation [30]. Yet, the subtle distinction between these early approaches and the native INDP paradigm is that they fail to directly deploy learning models on the data plane due to various hardware constraints. For example, the lack of support for floating-point arithmetic on P4 switches makes it significantly more difficult to execute model inference on the data plane than on general-purpose processors like CPUs and GPUs.

The community since then make substantial progress on realizing tree-based INDP [7,24,57,58,68,69,71], based on the insight that the decision making process in tree-models can be implemented using match-action tables on the programmable data plane. State-of-the-art (SOTA) in this regard is NetBeacon [71] which designs a novel ternary table encoding mechanism to efficiently deploy fairly large tree/forest models on programmable switches. Further, the recent art [43, 50–52] embrace neural networks by deploying binarized Multi-Layer Perceptron (MLP) on SmartNIC. Yet, the capacity of SmartNIC (*e.g.,* 2×40 GbE for Netronome Agilio CX [37]), which co-locates with an end-host, is several orders of magnitude smaller than the in-network programmable switches (*e.g.,* 6.4 Tbps for Barefoot Tofino 1 switch).

In this paper, we propose Brain-on-Switch (BoS) to advance state-of-the-art of INDP in two fundamental ways. First, BoS enables the use of Neural Network (NN) in INDP. NNs have several advantages over tree-based models for traffic analysis. For instance, Recurrent Neural Network (RNN), a

*Equal contribution. ✉ Corresponding author.

type of NN designed to work with sequential data, outperforms tree-models in both efficiency (*e.g.,* not requiring complicated feature computations on the fly, consuming fewer hardware resources to maintain per-flow state, etc.) and accuracy (especially when handling more complex tasks, such as *multi-class* traffic classification). Second, BoS is architecturally complete in the sense that it can accommodate full-precision and advanced models in INDP. Hardware limitations (*e.g.,* lack of floating-point number support) on switches force model binarization [2, 51, 58], which, unfortunately, reduces performance. Although prior art (*e.g.,* IIsy [68], [49]) mentioned the hybrid analysis concept of forwarding certain flows to large tree-based models deployed at the endpoint for reevaluation, they lack the fundamental design to precisely control the amount of such *escalated* flows processed off-switch. In contrast, BoS proposes a novel approach to accommodating advanced off-switch models (*e.g.,* transformer-based models) into INDP to improve the overall analysis performance, while ensuring that the vast majority of traffic (*e.g.,* over 95% of flows) is still analyzed at line-speed on the data plane.

Concretely, BoS has the following innovative designs:

(i) A novel binary RNN architecture that retains full-precision model weights during on-switch inference (*i.e.,* only activation functions are binarized), realized by encoding the complex layer forward propagation functions as match-action tables. Compared to the fully-binarized MLP model [51], our binary RNN exhibits substantial performance advantages.

(ii) A sliding-window based computation scheme to execute unlimited RNN time steps using limited forwarding stages on switches. BoS overcomes various switch hardware limitations in realizing the critical operations essential to this computation scheme, such as the read/write of a ring buffer like data structure, and an argmax like operation to make comprehensive inference decisions by aggregating the intermediate analysis results as a flow proceeds.

(iii) An analysis-escalation module to accommodate full-precision transformer-based models in BoS. The key design is two-fold: accurately identifying the flows for which on-switch analysis confidence is insufficient, and designing an Integrated Model Inference System (IMIS) to enable fast off-switch inference for escalated flows.

Contributions. The major contribution of this paper is the design, implementation and evaluation of BoS, the first INDP design that enables NN-driven traffic analysis at line-speed. We implement a prototype of BoS and evaluate it extensively using several use cases. The experimental results show that BoS outperforms SOTA in analysis accuracies by non-trivial margins, achieving up to ∼19% higher F1-scores than tree-based NetBeacon [71] and up to ∼40% higher than binary MLP based N3IC [51]. We further perform thorough system-level evaluations, demonstrating that BoS is scalable to handle high network loads (flow concurrency), attributing to the co-design of the on-switch binary RNN and off-switch IMIS. Finally, we evaluate hardware resource utilization by BoS.

2 Background and Motivation

Programmable Network Data Plane. The emerging of Protocol-Independent Switch Architecture (PISA) enables flexible data plane programmability, empowering fast innovations of networking designs. In PISA, the switching pipeline can be programmed via P4 [5], a domain-specific programming language. A PISA pipeline consists of a parser for header parsing, multiple match-action stages for header fields and metadata manipulating, and a deparser for header reassembling. In general, the actual packet processing logic is implemented using these match-action stages. PISA also supports components for stateful storage, such as registers.

Despite the programmability mentioned above, PISA has the following limitations. First, only simple operations like add, subtract, shift and bit-wise operations are supported, excluding floating numbers, multiplication, division and complex comparisons. Second, the resources (such as the number of stages, SRAM, TCAM) are limited. For instance, on Barefoot Tofino 1, each pipeline has 12 stages, 120 Mbit SRAM and 6.2 Mbit TCAM [71]. Finally, each register can only be accessed once through an atomic operation for each packet.

RNNs and Transformers. RNN [31] is designed to process sequence data of varying lengths by maintaining an internal state (*i.e.,* the hidden state). Specifically, given the input $x_t \in \mathbb{R}^m$ at time step t, the hidden state $h_t \in \mathbb{R}^n$ is calculated as $h_t = \tanh(W[x_t, h_{t-1}] + b)$, where W and b are trainable parameters. As h_t encodes the current input x_t and the historical information h_{t-1} at the same time, RNN can capture the relationships between the data points in a sequence. The algorithm used to calculate the hidden states is called a recurrent unit, and the two most popular recurrent units are LSTM [17] and Gated Recurrent Unit (GRU) [8].

Transformers [56] excel at modeling sequential data. Recently, several traffic analysis approaches [11, 26, 28, 59, 66] treat the bytes of packets as words or images, and introduce a variety of transformer-based models to achieve impressive traffic classification performance. In addition, the self-supervised pre-training paradigm used in transformer-based traffic analysis requires a small amount of labeled data.

Motivation. Our community make substantial progress [7, 24, 57, 58, 68, 69, 71] on realizing Intelligent Network Data Plane (INDP) by embedding decision tree models in the forwarding pipeline of programmable switches. Their key insight is that the decision making process in tree/forest models (*i.e.,* comparing a value to a threshold and then moving on to the next tree node until a leaf node is reached) is very similar to the match-action table paradigm used on the data plane. For instance, state-of-the-art NetBeacon [71] designs a novel coding algorithm to effectively represent multiple tree models using ternary matching tables.

We forecast a performance ceiling in further innovating tree-based INDP designs. Specifically, tree models often rely on advanced feature engineering (*i.e.,* extracting various types

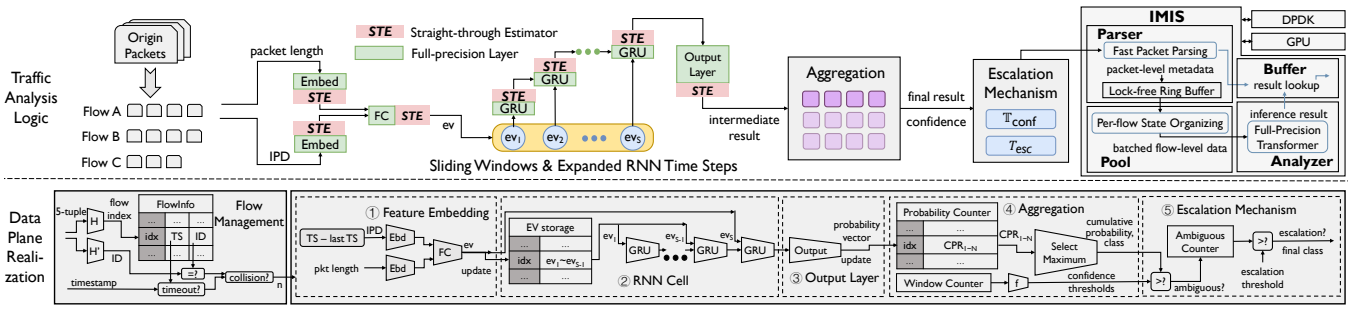


Figure 1: The BoS architecture enables NN-driven traffic analysis in INDP.

of statistics/properties/attributes from raw data) to boost accuracy. However, the features that are computable on the data plane at line-speed are fundamentally limited due to hardware constraints. For instance, flow features such as the s.t.d., frequency, and percentile of packet lengths are critical to tree models [40, 71]. Yet, computing these features is either impossible or difficult, often requiring ad-hoc tricks to estimate these statistics. For instance, prior art [71] estimates s.t.d. of packet lengths upon receiving certain packets (*i.e.,* the 2^k-th packet in each flow), indicating it can only execute inference at these locations. The limitation is obvious: an inference error obtained on the 2^k-th packet cannot be corrected until the arrival of the 2^{k+1}-th packet.

Design Goals. Therefore, philosophically, it is worth asking: can we expand the boundaries of INDP to a new type of learning models that is not limited by the availability of flow features on the data plane. In this paper, we address this research question concretely by enabling NN-driven traffic analysis in INDP. Unlike tree/forest models, many types of NNs (such as RNNs and transformers) that are designed to process sequential data can directly take raw network traffic data as input, eliminating the requirements of computing complex features on the data plane on the fly. However, the incorporation of NNs into INDP presents significant challenges. For instance, the recurrent computation scheme in RNN is fundamentally different from the match-action paradigm on the data plane, making it more difficult to realize on-switch RNN inference. Additionally, existing transformer-based traffic analysis approaches [11,26,28,59,66] simply treat network traffic as another form of sequential data, without constructing appropriate systems to analyze the network flows online while they are traversing the data plane.

To address these challenges, we architect BoS, the first NN-driven INDP system. A recent art N3IC [51] explores to deploy binary MLP models on SmartNIC, which is more computationally flexible, yet with much lower throughput than programmable switches. We focus on programmable switch based INDP in this paper (although we also compare the traffic analysis accuracy of BoS with N3IC in our evaluations). Concurrent with BoS, Broadcom unveils the early-stage development of their novel NN inference switching chip [6], underlining the significance of INDP.

3 Design Overview

We plot the architecture of BoS in Figure 1. The overarching traffic analysis logic in BoS centers around (i) a data plane friendly RNN inference architecture and (ii) a co-design with an Integrated Model Inference System (IMIS) to accommodate full-precision transformer-based traffic analysis models. The key designs toward hardware friendliness are two-fold: (i) realizing the online forward propagation of RNN layers via offline-trained input-output-mapping tables, and (ii) executing unlimited recurrence of RNN time steps via a sliding window mechanism that recurrently processes fixed-length packets segments. The key to co-design with IMIS is accurately identifying the flows that do not receive sufficient confidence from the on-switch analysis, and to only escalate these flows to the off-switch IMIS, so that BoS still processes the vast majority of traffic on-switch (*e.g.,* over 95% flows). Nevertheless, we optimize the system design of IMIS so that a single instance of IMIS can process ten million packets per second while maintaining low inference latencies (see § 7.3). This ensures the off-switch IMIS is unlikely to be the bottleneck of BoS.

4 Data Plane Friendly RNN Architecture

4.1 Raw Packet Sequences as Input Features

The on-switch RNN uses raw flow sequences as the input features, *i.e.,* the packet length sequence and the inter-packet delay (IPD) sequence. When a packet in the flow arrives at the switch, we extract its length and get IPD based on the subtraction of timestamps. Through feature embedding, these metadata are mapped into an embedding vector, which is stored in a sequence for subsequent model inference.

Using raw sequences as input features has several key advantages over using statistical flow features (such as the mean and s.t.d. of packet lengths). First, the availability of critical flow features is greatly limited on switch (explained in § 2). Second, storing per-flow statistical features on switch is expensive: NetBeacon [71] consumes 2.5x storage to store features as evaluated in § 7.2. Finally, feature engineering, without careful designs, could result in overfitting problem. For instance, we notice that some features (*e.g.,* the num-

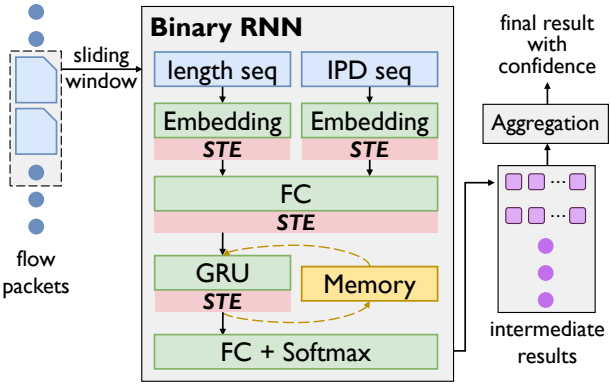


Figure 2: Data-plane-friendly binary RNN architecture.

ber of packets with packet size in $[48, 64)$) are heavily task-specific [3, 71], and features like port number can lead to overfitting on host configurations.

4.2 Binary RNN Architecture

As shown in Figure 2, architecturally, the binary RNN model in BoS consists of three building blocks: feature embedding, RNN cell and output layer. In the feature embedding, the length and IPD of each packet are passed through two different embedding layers, respectively, and then fed into a fully-connected layer to obtain an embedding vector. Taking the embedding vector sequence of per-flow packets as input, the RNN cell performs sequence analysis based on GRU [8]. In each time step of GRU calculation, the current embedding vector and the previous hidden state (*i.e.,* output activations of GRU) are used as input, and the output is used to update the hidden state. Finally, the hidden state of the last step is passed through the output layer (*i.e.,* a fully-connected layer with softmax) to obtain the analysis result.

Each GRU calculation contains 3 Hadamard product operations and 3 multiplications with nonlinear functions, which cannot be natively implemented on programmable switches due to hardware constraints. To cope with this challenge, we perform binarization on neural network activation functions to enable hardware-friendly model deployment. Specifically, we set all activation functions in the feature embedding and the RNN cell to Straight-Through Estimator (STE) [64]. STE performs a sign function in forward propagation, which makes all neural network activations +1 or -1. And in backward propagation, STE estimates the incoming gradient to be equal to the clipped outgoing gradient.

Prior art N3IC [51] performs binarization on both weights and activations of an MLP model, and then implements fully-connected layer forward propagation on the SmartNIC using XOR and customized population count (popcnt) operations. The popcnt operation, unfortunately, is not friendly to the switch pipeline: realizing a single popcnt operation for a 128-bit string takes 14 switch stages. Yet, one 128bit-to-64bit fully-connected layer in an MLP model requires 64 popcnt

Table 1: Binary RNN v.s. Binary MLP

Prior Work	Binary Activations	Full Precision Weights	Stage Consumption⋆	Model Accuracy†
Binary MLP (N3IC [51])	✔	✗	High	Low
Binary RNN	✔	✔	Low	High

⋆ Estimated if we were to implement the binary MLP on a programmable switch.
† See § 7.2 for quantitative results.

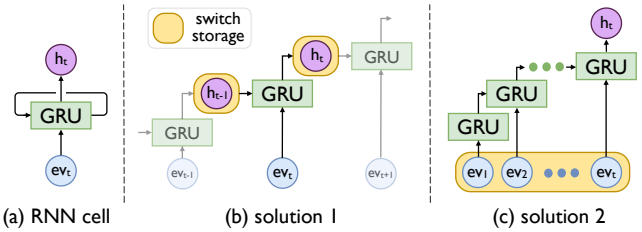


Figure 3: The design choices for RNN time steps.

operations. More crucially, as evaluated in § 7.2, full model binarization results in significant performance degradation. In Table 1, we summarize the key differences between our binary RNN and the binary MLP in N3IC [51].

4.3 Data Plane Native Model Inference

We now present the data plane native RNN inference.

Forward Propagation. The key to retain full precision model weights in our RNN models is to avoid direct computations of the layer forward propagation on the data plane. To this end, BoS realizes forward propagation based on match-action table lookup. Specifically, since all activations are binarized to +1 or -1, the input and output vectors of any neural network layer (*e.g.,* the embedding layer, FC layer and GRU layer in Figure 2) are essentially bit strings. Therefore, regardless of what computations are executed in a neural network layer, we can realize *equivalent input-output-relationship* by recording an enumerative mapping from input bit strings to output bit strings as a match-action table. Thus, in the online forward propagation through any layer, BoS uses input bit string as the key to match the output bit string stored in the corresponding table on switch. The caveat of this design is that the number of required entries N in each table is determined by the number of input bits, *i.e.,* $N = 2^{\text{input bit-length}}$. We recognize this constraint and demonstrate, via experiments (§ 7.2), that BoS can deploy efficient RNN models under this constraint.

RNN Time Steps. As shown in Figure 3(a), a straightforward way to implement RNN time steps is to store the RNN hidden state for each flow sequence. Upon packet arrival, we read the previous RNN hidden state, perform layer forward propagation, and then update the hidden state. Unfortunately, due to hardware constraint, each register can only be accessed once when a packet traverses the switching pipeline. Therefore, we need to expand RNN time steps in serial stages, as shown in Figure 3(b), where the read/write of hidden states spread across multiple stages. Alternatively, we can store a sequence

of embedding vectors corresponding to packet sequence of a flow, as shown in Figure 3(c). In online forward propagation, BoS calculates the embedding vector for each packet, updates the sequence in storage and executes all RNN time steps for this flow sequence in serial stages. We adopt this solution in the final prototype as it consumes fewer hardware resources.

Sliding Window Mechanism. Since the number of switch stages is limited, expanding the RNN time steps into serial stages would also limit the total number of time steps executable in our model. To address this problem, BoS designs a novel sliding window mechanism that can recurrently apply RNN inference on fixed-length flow segments. Therefore, the number of RNN time steps executable on a flow is no longer limited by switch stages. Specifically, in online traffic analysis, BoS uses a window with fixed-size S to extract a segment of S packets from the flow, executes S RNN time steps on this segment to obtain an intermediate inference result r_i, and swifts the window by one packet to obtain a new segment, and repeats the process. Therefore, BoS can continuously execute RNN time steps as the flow proceeds.

The key to fulfill the sliding window design is to properly aggregate these intermediate results. Specifically, upon receiving the j^{th} packet, suppose that the binary RNN has processed g full segments. Then the inference result for the j^{th} packet shall consider all the g intermediate inference results. In the case of multi-class traffic classification, one simple strategy is to select the majority class from these intermediate results. More crucially, we can co-design the aggregation algorithm with an off-switch module to improve the overall traffic analysis accuracy, as described below.

4.4 Analysis Escalation

Although BoS primarily relies on binary RNN to ensure line-speed traffic analysis, we still want to embrace full-precision and more advanced models (*e.g.,* transformers) to handle corner cases. For instance, in multi-class traffic classification, it is possible that none of the classes dominates (*e.g.,* the numbers of flow segments for different classes are close to each other). In this case, adopting the majority voting policy may reduce classification confidence.

To this end, BoS adopts an off-switch traffic analysis module co-located with the programmable data plane. We recognize two challenges in accommodating this analysis module. First, the aggregation algorithm must be carefully designed to ensure that it can accurately capture ambiguity, while avoiding consistently escalating flows (*i.e.,* only a small fraction of flows should be escalated to the co-located analysis module in order to preserve line-speed analysis for the vast majority of traffic). Second, the analysis module adopts a transformer-based model to improve accuracy. However, due to the complex computations required for inference, it is non-trivial to scale the online analysis throughput to maintain the high speed of network forwarding.

The Escalation Mechanism. To address the first challenge, we measure the *classification confidence* in the aggregation algorithm. Specifically, for each extracted packet segment in a flow, the binary RNN predicts an intermediate inference result, which is a vector of probabilities, one for each class. Suppose that upon receiving the j^{th} packet of the flow, the binary RNN has processed g packet segments for the flow (*i.e.,* the arrival of the j^{th} packet will form the $\{g+1\}^{\text{th}}$ segment). Then, our algorithm aggregates all $g+1$ intermediate inference results by accumulating the per-class prediction probabilities. The class with the largest cumulative probability is selected as the inference result for the j^{th} packet.

Whether a flow should be escalated is determined by the number of *ambiguous packets* in the flow. Upon receiving the j^{th} packet, suppose the largest cumulative probability among all classes is CPR_m and the total number of intermediate results is wincnt, then the classification confidence for the j^{th} packet is quantified as $\mathsf{CPR}_m/\mathsf{wincnt}$. The packet is considered ambiguous if its confidence is below a predefined *confidence threshold*. We use $\mathbb{T}_{\mathsf{conf}}$ to represent the vector of confidence thresholds, one for each class. The flow is escalated when the number of ambiguous packets in the flow exceeds a predefined *escalation threshold* T_{esc}.

$\mathbb{T}_{\mathsf{conf}}$ and T_{esc} are learned based on the distributions of the classification confidences of the training samples. Consider the example in Figure 4. For the VoIP class in the ISCXVPN2016 dataset (see detailed descriptions in § 7.1), we plot the CDF of the confidence scores for both correctly classified packets and misclassified packets. The confidence scores are quantized because they are eventually computed on the data plane (see § 5.2). An appropriate $\mathbb{T}_{\mathsf{conf}}$ should escalate as many misclassified packets as possible without affecting correctly classified packets. To this end, we design the following loss function to train our binary RNN.

Suppose p_i is the RNN's prediction probability for class i and y is the ground-truth class. The classic cross entropy (CE) loss is $\text{CE} = -\log(p_y)$. The CE loss solely considers to improve the model's ability to predict the correct class. Our loss is defined as $\mathcal{L}_1 = -(1-p_y)^\gamma \log(p_y) - \lambda \sum_{i \neq y} p_i^\gamma \log(1-p_i)$, which adds another item to explicitly negate the model's prediction on the non-ground-truth classes. The factor λ balances the two items, while the modulating factors $(1-p_y)^\gamma$ and p_i^γ down-weight easy samples and focus on hard samples, as proposed in the Focal Loss [27]. Intuitively, this loss enhances the confidence differences between misclassified and correctly classified packets by reducing $p_i (i \neq y)$ while retaining high p_y. Since our aggregation algorithm chooses the class with the largest accumulative probability, a simplified version of the above loss function is to only reduce the maximum prediction probability among all the non-ground-truth classes, *i.e.,* $\mathcal{L}_2 = -(1-p_y)^\gamma \log(p_y) - \lambda p_{\mathsf{false}}^\gamma \log(1-p_{\mathsf{false}})$, where p_{false} is the largest p_i among all the non-ground-truth classes. We thoroughly evaluate these loss functions in § 7.3.

Once $\mathbb{T}_{\mathsf{conf}}$ is determined, we set T_{esc} to control the amount

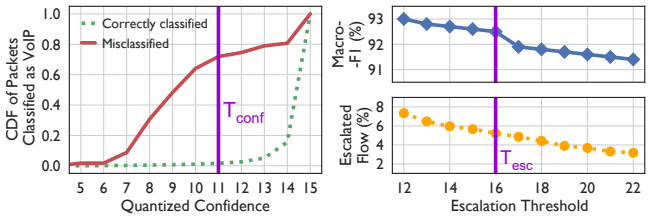


Figure 4: The selection of T_{conf} and T_{esc}.

of escalated flows. As shown in Figure 4, we select a T_{esc} to ensure that no more than 5% flows are escalated to the co-located analysis module for further analysis.

Integrated Model Inference System. To address the second challenge, we design an IMIS that enables fast online inference for a full-precision transformer-based model. As illustrated by Figure 1, IMIS orchestrates four types of stateful and single-threaded tasks (called *engines*) to realize a *non-blocking* traffic processing pipeline. We describe the transformer model training and the architecture of IMIS in § 6.

4.5 Integrated Analysis Logic

Algorithm 1 summarizes the complete logic of our online traffic analysis in BoS. Because our binary RNN model leverages flow-level data for inference, BoS designs a dedicated flow manager to store per-flow state. When a packet $\mathcal{P}$ is received, the flow manager first checks if per-flow state for $\mathcal{P}$ has already been allocated. If not, the flow manager allocates new per-flow storage for the packet. The packet then enters the normal flow-aware inference pipeline based on the retrieved flow state. Due to the limited capacity of the switch, when the flow manager cannot allocate storage for a new flow, BoS falls back to analyzing the packets of that flow using a tree model trained only using per-packet features (*e.g.,* packet length, TTL, Type of Service, TCP offset). This tree model is deployed on the data plane alongside our binary RNN model. The detailed design of the flow manager is deferred to § A.1.4.

We elaborate on one key design that has not been thoroughly discussed yet. In line 24, we periodically reset the window counter and per-class results every $\mathcal{K}$ packets. This effectively clears the contributions of very ancient flow segments (*i.e.,* more than $\mathcal{K}+1$ packets apart) when aggregating the intermediate inference results. This design rationale is that if we obtain a sub-flow f_{sub} by extracting a continuous and sufficiently long sequences of packets (starting from any position) from a flow f, it is very likely that f_{sub} and f are classified as the same class. Thus, clearing the results of very remote segments will not affect traffic analysis results. On the contrary, without periodical reset, the per-class results CPR would be consistently accumulated. To prevent buffer overflow, we need to allocate more bits to store CPR, which, unfortunately, results in significant hardware resource consumption (see § 5.2). Note that the periodical reset does not clear the embedding vectors for the previous $S-1$ packets.

Algorithm 1: Integrated Traffic Analysis Logic

Define : WIN$[0\ldots S-1]$ sliding window; N No. of classes; CPR$[0\ldots N-1]$ per-class results; $\mathbb{T}_{conf}[0\ldots N-1]$ the per-class confidence threshold; T_{esc} the escalation threshold

1 **if** FlowManager(packet $\mathcal{P}$) *fails* **then**
2 Fall back to use the per-packet model, and exit
3 Retrieve the flow state for $\mathcal{P}$
4 **if** $\mathcal{P}$ *is matched by the* EscTable **then**
5 Forward $\mathcal{P}$ to IMIS, and exit ▷ Escalated flows
6 pktcnt $\leftarrow$ pktcnt $+1$ ▷ Count packets
7 $ev \leftarrow$ **FeatureEmbedding**($\mathcal{P}$.length, $\mathcal{P}$.IPD)
8 WIN[pktcnt % S] $\leftarrow ev$ ▷ Slide the window
9 **if** pktcnt $< S$ **then** ▷ The first $S-1$ packets
10 Pre-analysis packet handling ▷ See § A.1.6
11 **else**
12 $h \leftarrow \vec{0}$
13 **for** $i \leftarrow 1$ **to** S **do** ▷ RNN time steps
14 $ev_i \leftarrow$ WIN[(pktcnt $+i$) % S]
15 $h \leftarrow$ **RNNCell**(ev_i, h)
16 $PR \leftarrow$ **OutputLayer**(h) ▷ Intermediate result: a probability vector
17 CPR $\leftarrow$ CPR $+ PR$
18 Class $\leftarrow$ argmax(CPR) ▷ Measure confidence
19 wincnt $\leftarrow$ wincnt $+1$ ▷ No. of windows
20 **if** CPR[Class] $< \mathbb{T}_{conf}$[Class] $*$ wincnt **then**
21 esccnt $\leftarrow$ esccnt $+1$ ▷ Ambiguous packets
22 **if** esccnt $\geq T_{esc}$ **then**
23 Initiate escalated analysis for subsequent packets
24 **if** pktcnt % $\mathcal{K} = 0$ **then** Reset(wincnt, CPR)

5 Model Realization on the Data Plane

5.1 Embedding Vector Storage and Retrieval

As shown in Figure 3(c), the RNN cell takes the embedding vectors of the packets in each sliding window (with length S) as input. Consider the flow segment starting with the k^{th} packet and ending with the $\{k+S-1\}^{\text{th}}$ packet. Upon arrival of the $\{k+S-1\}^{\text{th}}$ packet, the current flow segment is full and is ready to execute all S RNN steps simultaneously, *i.e.,* using the embedding vector of each packet as the key to obtain the matched output in each GRU table (S tables in total). Thus, before a segment is full, we need to temporally hold the embedding vectors for prior $S-1$ packets. We design a ring buffer with $S-1$ bins (registers) to store embedding vectors. In particular, the k^{th} packet in the segment is stored in the $\{k \% (S-1)+1\}^{\text{th}}$ bin (indexed from 1) of the ring. The embedding vector of the $\{k+S-1\}^{\text{th}}$ packet eventually takes the bin occupied by the k^{th} packet, which will be out-of-scope upon the arrival of the $\{k+S\}^{\text{th}}$ packet. The second benefit of the ring buffer design is that all bins are mutually independent, so that they can be accessed in parallel.

The efficient storage of embedding vectors leads to a chal-

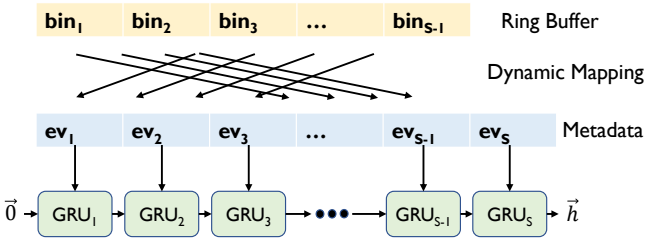


Figure 5: Storage and retrieval of embedding vectors.

lenge in reading these vectors. Specifically, given a flow segment, the embedding vector of the first packet in this segment is not always stored in the first bin of the ring buffer. Therefore, it is incorrect to statically align the ring buffer with GRU tables, *i.e.,* using the value stored in the k^{th} bin as the input key of the k^{th} GRU table. Instead, the input of the i^{th} GRU (for $i \in [1, S-1]$) should be read from the $\{(k-S+i)\%(S-1)+1\}^{\text{th}}$ bin. However, when declaring a lookup table for GRU on switch, the storage locations of its keys must be static and predetermined. Thus, to realize the above dynamic mapping, we need to first read values from the ring buffer to several intermediate variables (called metadata), and then dispatch the metadata to the proper GRU tables, as shown in Figure 5.

5.2 Intermediate Results Aggregation

As described in § 4.4, the key operation in our RNN inference is to select the largest cumulative probability from all intermediate inference results, *i.e.,* executing an argmax operation.

Ternary-Matching Based Design. Argmax is not a primitive available on the switch. We realize argmax based on an efficient data plane design of number comparison. Intuitively, number comparison can be accomplished by either conditional statements or exact-matching table matching. Neither of them, however, is scalable (see § A.1.1).

In BoS, we propose a scalable ternary-matching based design. Suppose argmax compares n numbers each with m bits. The key of a table entry consists n segments, each with m ternary bits (*i.e.,* 0, 1, or $*$). The value represents the winner (*i.e.,* the largest number). Starting from the most significant bit (MSB), to generate the l^{th} bit for each key segment, there are 2^n possible cases. Consider a case $\mathcal{C}_{(l,k)}$ where the l^{th} bits of the first k ($k \in [1, n\text{-}1]$) segments are 1 and the l^{th} bits of the remaining segments are 0. Clearly, the segments whose l^{th} bits are 0 will not be the winner, so that we can stop further enumerating the lower bits (*i.e.,* the $\{l+1\}^{\text{th}}, \{l+2\}^{\text{th}}, \ldots, m^{\text{th}}$ bits) for these segments. Thus, among all $2^{(m-l)\cdot n}$ sub-cases of the case $\mathcal{C}_{(l,k)}$, we do not need further enumerations for $2^{(m-l)\cdot k}$ of them, achieving a $2^{(m-l)\cdot(n-k)}$ reduction ratio. Take $\mathcal{C}_{(1,1)}$ as an example: it represents the case where the first (*i.e.,* the most significant bit, MSB) of the first segment is 1 and the MSBs for other remaining segments are 0. Thus, all $2^{(m-1)n}$ sub-cases for $\mathcal{C}_{(1,1)}$ have clear winners and are collapsed into one key (*e.g.,* $1***, 0***, 0***$ if n=3, m=4).

```
Inputs:
    n: number of keys; m: bit width of each key
1:  Procedure Generate(n, m):
2:      T = {1...n}  ▷ initial input, all the numbers can be the winner
3:      ▷ For explanation purpose, entry is represented as a 2-D array
4:      entry = array[1..n][1..m]  ▷ array of ternary bits (0, 1, *)
5:      Work(T, 1)
6:  Procedure Work(S, L):  ▷ S: possible winners in this iteration
7:      for num ∈ T\S ⇒ entry[num][L] = '*'  ▷ cases cannot win
8:      if L = m ⇒ Output(S), return  ▷ base case: last bit
9:      for φ ⊂ S' ⊂ S:  ▷ cases similar to C_(L,|S'|)
10:         for num ∈ S\S' ⇒ entry[num][L] = '0'
11:         for num ∈ S' ⇒ entry[num][L] = '1'
12:         Work(S', L+1)  ▷ Recursive resolve
13:     for num ∈ S ⇒ entry[num][L] = '*'  ▷ case C_(L,0) & C_(L,|S|)
14:     Work(S, L+1)  ▷ Recursive resolve
15: Procedure Output(S):  ▷ S: possible winners in this iteration
16:     a = list(S) in INCREASING order  ▷ indexed from 1
17:     for i = len(a) downto 2:  ▷ winning case for a[i≥2]
18:         for k ∈ [1..i-1] ⇒ entry[a[k]][m] = '0'
19:         entry[a[i]][m] = '1'
20:         for k ∈ [i+1..len(a)] ⇒ entry[a[k]][m] = '*'
21:         install(entry, winner=a[i])
22:     for num ∈ S ⇒ entry[num][m] = '*'  ▷ winning case for a[1]
23:     install(entry, winner=a[1])
```

Figure 6: The procedure to generate a ternary-matching table to realize argmax on the data plane.

Based on the above protocol, we derive the number of required table entries $F(n,m)$ as (see details in § A.1.2).

$$F(n,m) = 2 * F(n,m-1) + \sum_{i=1}^{n-1} \binom{n}{i} F(i,m-1) \quad \text{for} \quad n,m \geq 2 \qquad (1)$$
$$F(n,1) = 2^n \text{ for } n \geq 2; \quad F(1,m) = 1 \text{ for } m \geq 1.$$

Further Optimizations. We make two subsequent optimizations to further reduce $F(n,m)$. First, the two special cases $\mathcal{C}_{(l,n)}$ (*i.e.,* the l^{th} bits in all n segments are 1) and $\mathcal{C}_{(l,0)}$ (*i.e.,* the l^{th} bits in all n segments are 0) can be further merged. Specifically, for all 2^{m-l} sub-cases of $\mathcal{C}_{(l,n)}$, their winners remain the same if we modify the l^{th} bits of all n segments to 0; and similarly for 2^{m-l} sub-cases of $\mathcal{C}_{(l,0)}$, their winners remain the same if we modify the l^{th} bits of all the n segments to 1. Thus, $\mathcal{C}_{(l,0)}$ and $\mathcal{C}_{(l,n)}$ can be merged by modifying the l^{th} bit as an wildcard asterisk in each segment. We handle this merged case lastly in the current enumeration of the l^{th} bit (see lines 13 and 14 in Figure 6), so that these wildcard asterisks will not interfere with previous cases with higher priority (see lines 9 to 12 in Figure 6). With this optimization, $F(n,m)$ is reduced as $F(n,m) = F(n,m-1) + \sum_{i=1}^{n-1} \binom{n}{i} F(i,m-1)$.

The second optimization is reducing the base case $F(n,1)$. By reversely encoding the one-bit number comparison (see Figure 7), $F(n,1)$ is reduced to n from 2^n. Combining both optimizations, we obtain $F(n,m) = nm^{n-1}$.

$segment_1$	$segment_2$	···	$segment_{n-1}$	$segment_n$	Winner
x0	x0	···	x0	x1	n^{th} number
x0	x0	···	x1	x*	$\{n-1\}^{th}$ number
...	...	...	...	...	...
x0	x1	···	x*	x*	2^{nd} number
x*	x*	···	x*	x*	1^{st} number

Figure 7: The reverse encoding for $F(n,1)$.

6 Implementation

Our BoS prototype[1] includes: 1500 lines of Python code for model training, 1900 lines of P4 code for on-switch RNN, and 3300 lines of C++ code for IMIS. To evaluate the prototype, an additional 1600 lines of code are developed.

Model Training. We train a binary RNN to analyze flow segments extracted by the sliding window. Given the window size S and a flow sample $(\mathcal{P}_1, \mathcal{P}_2, \dots)$ in the training dataset, we slice this flow into all possible packets segments (*e.g.,* consecutive S packets like $(\mathcal{P}_1, \dots, \mathcal{P}_S)$ and $(\mathcal{P}_2, \dots, \mathcal{P}_{S+1})$) where the label of each segment is the flow label. For each segment, we use its packet length sequence and IPD sequence as inputs, and train the binary RNN to predict its label correctly. Recall that our binary RNN outputs vector of probabilities, one for each class. The training process is to maximize the prediction probability on the ground-truth class.

We use YaTC [66], a recent masked autoencoder (MAE) [16] based traffic transformer with multi-level flow representation, in IMIS to analyze escalated flows. YaTC only uses the first 5 packets of a flow for analysis. For each packet, it extracts the first 80 header bytes and 240 payload bytes as inputs. We first determine the two thresholds in § 4.4 to collect the escalated flows in the training set, and then fine-tune the pre-trained YaTC model [66] to obtain our final model.

On-Switch RNN Implementation. We implement a prototype on our Tofino 1 programmable switch. The top part of Figure 8 shows the workflow of all the components in our prototype. The left-bottom table in Figure 8 lists the hyper-parameters of our prototype, and the right-bottom table lists the detailed per-stage arrangement of our components. Due to space constraints, we defer the detailed description of our prototype to § A.2.1. Although the hardware resources on the Tofino 1 are very limited (*e.g.,* only 12 stages), we manage to implement a prototype that supports all four traffic analysis tasks evaluated in § 7.1. The on-switch RNN is programmable in runtime via the control plane (see § A.3).

IMIS Implementation. The core design of IMIS is a non-blocking traffic processing pipeline. As illustrated in Figure 1, architecturally, IMIS is designed around stateful, single-threaded tasks, which we call *engines*. The parser engine uses DPDK [18] (version 20.11) APIs to consistently collect the packet bytes from the escalated traffic; the pool engine takes the stream data as input and organizes it into per-flow state; the analyzer engine calls the pool engine to collect a batch of fresh per-flow data, and uses CUDA (version 11.7) [38] to interact with an auxiliary GPU to accelerate model inference; and the buffer engine stops packets without inference results to wait in memory, and sends those who have inference results to NIC. The pool engine is the key to dynamically coordinate the speeds of the parser engine and analyzer engine, thus achieving a non-blocking packet processing pipeline. The detailed architecture of the IMIS system is deferred to § A.2.2.

[1] Available at https://github.com/InspiringGroup-Lab/Brain-on-Switch

7 Evaluation

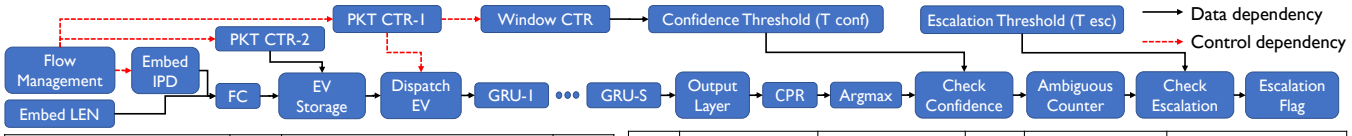
7.1 Experiment Setup

Testbed Setup. We deploy our binary RNN model using one pipe of a Barefoot Tofino 1 programmable switch. One server generates network traffic to an inbound port of the switch based on the pcap files we created for various traffic analysis tasks and traffic loads. Each flow is either analyzed by the on-switch RNN or redirected from one specific switch port to an off-switch server that deploys IMIS. For the flows analyzed on-switch, we develop a dedicated on-switch module to collect their analysis statistics online. Scaling the on-switch analysis of BoS beyond a single pipe of the switch is feasible given proper flow management. We discuss this in § A.3.

Tasks. We evaluate BoS using the following four tasks. (i) Encrypted traffic classification on VPN: this task classifies network traffic encrypted by VPNs. We use the ISCXVPN2016 [12] dataset, a six-class classification task (Email, Chat, Streaming, FTP, VoIP, P2P). (ii) Botnet traffic classification: this task classifies botnet traffic collected from the IoT systems. We use the BOTIOT [22] dataset, a four-class classification task (Data Exfiltration, Key Logging, OS Scan, Service Scan). (iii) Behavioral analysis of IoT devices: this task classifies traffic generated by IoT devices in different working states. We use the CICIOT2022 [10] dataset, a three-class classification task (Power, Idle, Interact). (iv) P2P application fingerprinting: this task classifies network traffic generated by P2P applications. We use the PeerRush dataset [40], a three-class classification task (eMule, uTorrent, and Vuze). We supplement additional details regarding the processing of these datasets in § A.4.

Network Load. We would like to evaluate BoS under different network loads. Similar to prior art [51, 61, 71], we use the number of new flows arrived in each second to represent the network load. Specifically, for each testing flow in a task, we extract its raw packets from the pcap file while preserving the inter-packet delays. Given the total number of flows in this task, and a desired network load, we calculate the total time period required to replay these flows, and then uniformly release these flows within this period. If the period is too short, we replay these flows multiple times in a loop to create consistent loads throughout our test.

The actual network load varies in different deployment. We make load estimations based on prior measurements. In 2015, Meta [42] reported that its external-facing web server generates 500 new flows per second (median). Meanwhile,



hyper-parameter	value	hyper-parameter	value
Window Size (S)	8	Reset Period of Window Counter (K)	128
No. of Classes (N)	6	Bit Width of Intermediate Probability	4
Bit Width of Embedded LEN	10	Bit Width of Cumulative Probability	11
Bit Width of Embedded IPD	8	Bit Width of ID	32
Bit Width of Embedding Vector	6	Bit Width of TS	32
Bit Width of Hidden State	9	Flow Capacity	65536

stage	ingress	egress	stage	ingress	egress
0	calculate ID, idx embed pkt length	GRU-5 window_counter	5	FC escalation_flag	CPR-4,5,6 u ← argmax(CPR-1,2,3)
1	FlowInfo	GRU-6	6	bin-4,5,6,7	v ← argmax(CPR-4,5,6)
2	last_TS pkt_counter-1,2	GRU-7 calculate threshold	7	bin-1,2,3	argmax(u,v)
			8	dispatch ev	ambiguous_counter
3	calculate IPD	Output ◦ GRU-8	9	GRU-2 ◦ GRU-1	set mirror
4	embed IPD	CPR-1,2,3	10~11	GRU-3, GRU-4	

Figure 8: The breakdown of our on-switch RNN implementation on a Tofino programmable switch.

Table 2: Experimental settings.

Datasets (Tasks)	ISCXVPN 2016	BOT IOT	CICIOT 2022	Peer Rush
Training Flows	7801	7835	5332	30770
Testing Flows	1951	1961	1335	7694
Classes	6	4	3	3
Class Ratio⋆	2:6:1:5:9:3	1:1:4:19	1:4:1	2:1:1
Best Loss	$\mathcal{L}_1$	$\mathcal{L}_1$	$\mathcal{L}_2$	$\mathcal{L}_1$
λ, γ	0.8, 0	0.5, 0.5	3, 1	1, 0
Optimizer	AdamW	AdamW	AdamW	AdamW
Learning Rate	0.01	0.005	0.005	0.005
RNN Hidden States†	9 bits	8 bits	6 bits	5 bits
Per-packet Model Acc.	0.596	0.327	0.759	0.684
Network Load	Low	Normal	High	Scaling
No. of flows / s	1000	2000	4000	up to 7.8M

⋆ See § A.4 for the accurate numbers of flows in each class.
† We evaluate BoS under different binary RNN model sizes in § A.6.

CISCO [9] measures that Internet traffic grows 3-fold from 2016 to 2021. Combining these measurements, we estimate that 2000 new flows per second are a reasonable network load that BoS may face in practice. In our scaling test (see § 7.3), we stress test BoS with up to 450,000 new flows per second on our testbed (a 225x increase from the normal load, and 30-300x over NetBeacon [71]).

Metrics. We use packet-level macro-F1 (the average of F1-score for different classes) as the accuracy metric, and further report the breakdown of the Precision / Recall of each class.

7.2 End-to-end Performance

In this section, we report the end-to-end performance of BoS for different tasks. The main experimental settings are summarized in Table 2. In § 7.3, we evaluate BoS under a variety of settings. We also compare BoS with two recent art NetBeacon [71] and N3IC [51]. N3IC deploys the binary MLP on a SmartNIC. For fair comparison, we simulate the switch-side traffic management logic and the binary MLP inference in software to obtain the traffic analysis results for N3IC. The detailed descriptions about the reproduced versions of the two art are given in § A.5.

Accuracy. We summarize the analysis accuracy results in Table 3. Across all evaluated tasks, BoS achieves significantly better performance than NetBeacon and N3IC, with an average F1-score improvement of 0.13 and 0.31, respectively. On more challenging tasks with more classification classes, the improvement is even greater, up to 0.19 and 0.42, respectively. We observe the binary MLP performs the worst because the accuracy loss caused by binarizing all model weights is significant. In fact, on the ISCXVPN2016 and CICIOT2022 datasets, the F1-scores of N3IC are even lower than these of our fallback tree-based model (0.596/0.759). Constrained by the availability of flow features, NetBeacon can only execute model inference at discrete locations. Thus, an inference error affects all its subsequent packets until it is corrected by the next inference point. This fundamentally limits its F1-scores, especially for more difficult tasks. In contrast, BoS retains full-precision model weights in the on-switch RNN model and continuously produces fresh inference results as a flow proceeds. Together with the co-located IMIS, BoS produces more accurate analysis results than existing arts, achieving over 0.920 F1-score in all tasks. We observe very minor declines of F1-scores in BoS as the network load increases, demonstrating the effectiveness of our flow management (note that we use the same flow management module for other two systems as well). The minor accuracy loss is because a small fraction of flows (*e.g.,* 2.77%/1.43%/2.05%/5.22% in the normal network load case) fall back to using the per-packet model.

Hardware Resource Utilization. We report the stateful SRAM and stateless SRAM/TCAM usage by BoS on the programmable switch in Table 4. The stateful SRAMs are consumed to maintain per-flow states, which mainly consist of the flow management information (*e.g.,* TrueID and timestamp, see § A.1.4), the embedding vectors (EV) for binary RNN inference, and the cumulative probability counter (CPR) for each class. In our prototype, the hardware consumption for the first two parts is task-irrelevant, and one task uses roughly 8.85% of SRAM. The consumption for the last part depends on the number of classification classes in a task, and the four tasks use roughly 5.63%/3.75%/2.81%/2.81% of SRAM, respectively. The embedding vectors stored for each flow take $8 \times (S-1) + 8$ bits (64 bits in our prototype). Compared with existing approaches [51, 71] that require online feature com-

Table 3: Analysis accuracy for BoS and other two closely related art.

Methods	BoS			NetBeacon [71] (Tree-based Models)			N3IC [51] (Binary MLP)		
Network Load	Low	Normal	High	Low	Normal	High	Low	Normal	High
Encrypted Traffic Classification on VPN (ISCXVPN2016)									
Email	0.935 / 0.933	0.936 / 0.925	0.933 / 0.923	0.309 / 0.514	0.315 / 0.524	0.320 / 0.525	0.347 / 0.326	0.354 / 0.339	0.367 / 0.350
Chat	0.903 / 0.818	0.902 / 0.818	0.901 / 0.814	0.739 / 0.935	0.739 / 0.933	0.742 / 0.925	0.336 / 0.655	0.336 / 0.654	0.342 / 0.656
Streaming	0.926 / 0.941	0.926 / 0.939	0.926 / 0.910	0.963 / 0.919	0.962 / 0.904	0.962 / 0.874	0.741 / 0.608	0.742 / 0.603	0.743 / 0.581
FTP	0.973 / 0.928	0.973 / 0.926	0.973 / 0.922	0.946 / 0.659	0.946 / 0.655	0.947 / 0.654	0.563 / 0.396	0.567 / 0.396	0.575 / 0.397
VoIP	0.968 / 0.958	0.968 / 0.958	0.968 / 0.957	0.938 / 0.882	0.939 / 0.881	0.939 / 0.882	0.883 / 0.783	0.884 / 0.782	0.886 / 0.787
P2P	0.905 / 0.927	0.903 / 0.928	0.876 / 0.930	0.810 / 0.959	0.798 / 0.959	0.778 / 0.960	0.578 / 0.739	0.577 / 0.742	0.565 / 0.748
Macro-F1	0.926	0.925	0.919	0.786	0.784	0.780	0.565	0.567	0.568
Botnet Traffic Classification on IoT (BOTIOT)									
Data Exfiltration	0.964 / 0.974	0.951 / 0.973	0.899 / 0.971	0.691 / 0.845	0.684 / 0.847	0.658 / 0.848	0.514 / 0.879	0.508 / 0.881	0.506 / 0.879
Key Logging	0.960 / 0.946	0.961 / 0.962	0.959 / 0.902	0.921 / 0.425	0.921 / 0.419	0.918 / 0.399	0.055 / 0.033	0.058 / 0.033	0.052 / 0.031
OS Scan	0.996 / 0.996	0.995 / 0.989	0.995 / 0.966	0.838 / 0.963	0.841 / 0.963	0.844 / 0.945	0.831 / 0.693	0.830 / 0.677	0.831 / 0.672
Service Scan	0.993 / 0.992	0.986 / 0.973	0.979 / 0.978	0.928 / 0.876	0.927 / 0.870	0.917 / 0.858	0.845 / 0.663	0.830 / 0.664	0.840 / 0.663
Macro-F1	0.978	0.974	0.955	0.785	0.782	0.769	0.547	0.542	0.541
Behavioral Analysis of IoT Devices (CICIOT2022)									
Power	0.926 / 0.887	0.924 / 0.882	0.921 / 0.882	0.819 / 0.726	0.820 / 0.724	0.817 / 0.724	0.639 / 0.750	0.640 / 0.750	0.640 / 0.748
Idle	0.922 / 0.943	0.921 / 0.942	0.918 / 0.941	0.810 / 0.938	0.808 / 0.938	0.806 / 0.936	0.618 / 0.640	0.620 / 0.642	0.622 / 0.646
Interact	0.934 / 0.946	0.934 / 0.948	0.934 / 0.943	0.871 / 0.786	0.873 / 0.786	0.872 / 0.784	0.651 / 0.504	0.655 / 0.506	0.661 / 0.510
Macro-F1	0.926	0.925	0.923	0.822	0.821	0.820	0.629	0.631	0.633
P2P Application Fingerprinting (PeerRush)									
eMule	0.943 / 0.949	0.918 / 0.949	0.898 / 0.950	0.846 / 0.954	0.821 / 0.955	0.805 / 0.954	0.734 / 0.866	0.730 / 0.867	0.723 / 0.875
uTorrent	0.949 / 0.924	0.950 / 0.912	0.941 / 0.894	0.882 / 0.870	0.885 / 0.858	0.885 / 0.831	0.734 / 0.789	0.735 / 0.790	0.738 / 0.783
Vuze	0.946 / 0.962	0.945 / 0.947	0.941 / 0.930	0.910 / 0.810	0.907 / 0.790	0.904 / 0.793	0.821 / 0.626	0.826 / 0.622	0.826 / 0.616
Macro-F1	0.945	0.937	0.925	0.877	0.866	0.858	0.755	0.755	0.752

Table 4: Hardware resource utilization.

	Datasets (Tasks)	ISCXVPN 2016	BOT IOT	CICIOT 2022	Peer Rush
SRAM	Flow Info. (stateful)	5.21%	5.21%	5.21%	5.21%
	EV (stateful)	3.65%	3.65%	3.65%	3.65%
	CPR (stateful)	5.63%	3.75%	2.81%	2.81%
	FE (stateless)	2.19%	2.19%	2.19%	2.19%
	GRU (stateless)	3.02%	1.56%	0.73%	0.73%
	Total⋆	23.44%	20.10%	18.33%	18.33%
TCAM	Argmax (Total)	1.74%	1.04%	0.69%	0.69%

⋆ Including other components not listed, *e.g.,* packet counters for each flow.

putation, their per-flow storage consumption depends on the used flow features. For instance, NetBeacon [71] engineers 7 important features for the P2P application fingerprinting task, which consumes roughly 150 bits.

The stateless SRAM is used to implement the lookup tables for feature embedding (FE) and GRU layers in our binary RNN. Specifically, the SRAM consumption of GRU layers depends on the number of bits used for storing RNN hidden states. Using the default bitwidth in Table 2, the four tasks use roughly 3.02%/1.56%/0.73%/0.73% of SRAM, respectively, for GRU layers. Additionally, each task uses 2.19% of SRAM for feature embedding. In total, the four tasks use 23.44%/20.10%/18.33%/18.33% of SRAM, respectively.

BoS uses TCAM to implement the argmax operation. Compared with NetBeacon [71], BoS consumes SRAM of similar size and 20x less TCAM (note that the ternary matching in TCAM is 6x more expensive than exact matching with SRAM, in terms of required silicon resources [51]).

7.3 BoS Deep Dive

Analysis Escalation. In this segment, we study the trade-off between the amount of escalated flows and the overall macro-F1, demonstrating that our loss functions defined in § 4.4 achieve a better trade-off than the classic cross entropy loss. As described in § 4.4, the escalation threshold T_{esc} controls the amount of escalated flows. Using the setting in Table 2, we train the binary RNN with our losses and cross entropy loss, respectively, and measure the overall macro-F1 with different amount of escalated flows under the normal network load (2000 flows/s). The results are plotted in Figure 9, and the best parameters (λ, γ) of our losses in each task are presented. We make the following key observations. (i) Regardless of the used loss functions, the overall macro-F1 scores for all tasks improve as the percentage of escalated flows increases from 0% to 5%. This demonstrates the necessity for accommodating the off-switch analysis model to compensate for on-switch analysis. (ii) For the same amount of escalated flows, our losses outperform the cross entropy loss by significant margins across all tasks. This shows that our loss designs can more effectively identify the ambiguous packets that require additional reevaluation. This is crucial to improve the overall system performance without consistently escalating flows. (iii) The performance of our two losses $\mathcal{L}_1$ and $\mathcal{L}_2$ is task-dependent. In general, $\mathcal{L}_1$ outperforms $\mathcal{L}_2$ in three tasks, yet $\mathcal{L}_2$ requires less training epochs to converge.

System Performance of IMIS. In this segment, we stress test the performance of off-switch IMIS upon a burst of concurrent

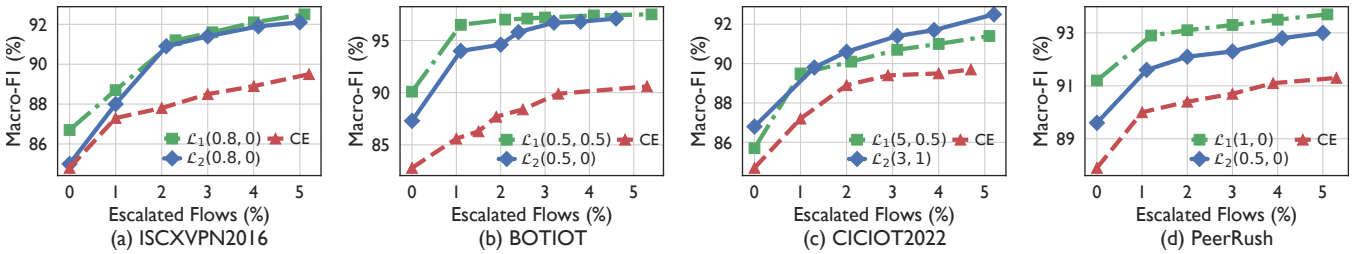


Figure 9: [Testbed] The trade-off between percentage of escalated flows and the overall accuracy.

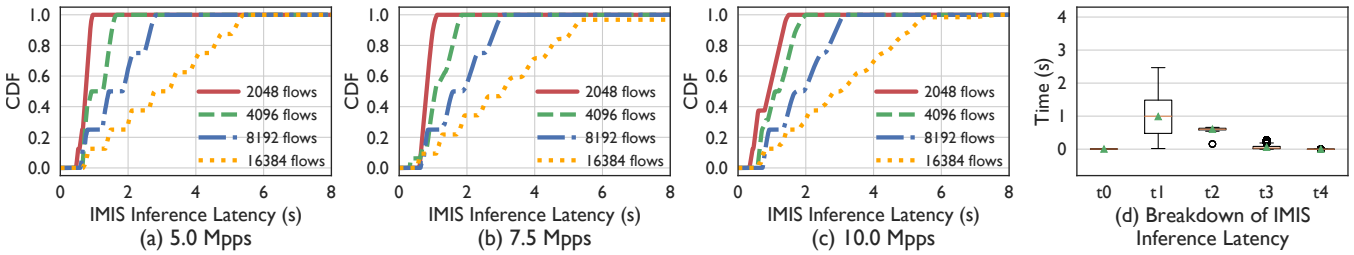


Figure 10: [Testbed] The inference throughput and latency of the off-switch IMIS.

flows. We run the IMIS with 8 parallel analysis modules. We evaluate four different levels of flow concurrency (2048, 4096, 8192, and 16384 flows) with three different aggregate inbound rates (5.0, 7.5 and 10.0 million packets per second). The complete inference pipeline for a packet $\mathcal{P}$ in IMIS has six phases: (1) $\mathcal{P}$ is fetched from the NIC by the parser engine; (2) its metadata is organized by the pool engine; (3) its metadata is sent to the analyzer engine; (4) the analyzer engine generates the inference result; (5) the result is collected by the buffer engine; (6) $\mathcal{P}$ is dispatched to NIC by the buffer engine.

The transformer model in IMIS performs inference on the first five packets of each flow. Given that the average length of the escalated flows in each tasks is 801, 255, 167, and 138 packets, respectively, the vast majority of packets in these escalated flows are directly forwarded to the buffer engine after being collected from the NIC, experiencing very minor latency (less than 1ms). In the following, we only consider the latency for the packets that traverse the entire inference pipeline. The CDFs of the end-to-end latencies are plotted in Figures 10(a) to (c). When the number of concurrent flows is below 4096, the maximum end-to-end latency imposed by IMIS is less than 2 seconds even for 10.0 Mpps inbound rate (equivalently 41 Gbps as the packet sizes we send are 512 B). Considering that BoS typically escalates less than 5% of flows, the flow concurrency levels experienced by the IMIS are expected to be low in most deployments. In Figure 10(d), we further report the breakdown of the end-to-end latency (*i.e.,* the time intervals between two consecutive phases in the inference pipeline) under 8192 concurrent flows and an inbound rate of 5.0 Mpps. We observe that the major latency occurs between the second and third phase, when the packets are waiting to be collected by the analyzer engine. The net inference time spent in the analyzer engine is about 0.6 s.

Scaling Test. We stress test BoS in high-throughput scenarios with high flow concurrency and high flow throughput. Specifically, because all the original network traces are collected in low bandwidth networks (*e.g.,* tens of Mbps), we create high-throughput network traces by concurrently packaging a large number of flows (while ensuring each flow has a unique identifier) and accelerating the packet replay speeds (by reducing the inter-packet delays). Then we replay these pcap files to generate traffic on our testbed. Figure 11 presents the scaling test results, where we progressively increase the flow concurrency to saturate the physical capacity of the NIC on our traffic generator. The results demonstrate that BoS can comfortably handle this level of scale, as the macro-F1 scores remain nearly identical compared to the results in Table 3.

To evaluate BoS at even larger scales, we build a simulator to emulate the entire workflow of BoS. The accuracy of the simulator is validated by replicating the experimental settings of Table 3 and Figure 11. The accuracy results obtained through the simulation are almost the same as those collected from our testbed. We subsequently employ the simulator to explore significantly larger scales, progressively increasing flow concurrency to up to 7.8 million flows per second, and the aggregate throughput to over 1.6 Tbps. The results depicted in Figure 12 reveal a sublinear decline in the macro-F1 scores of BoS, culminating in a $\sim$11.6% reduction at the largest scale.

Fallback Alternative. The default handling of the flows without dedicated per-flow storage is to analyze their packets using a tree-model trained trained only on per-packet features (see § A.1.5). Alternatively, a subset of the flows without dedicated per-flow storage can be forwarded to a new instance of off-switch IMIS dedicated to handling these flows. In Figure 11 and 12, we report the macro-F1 when forwarding a certain percentage of flows without per-flow storage to a dedicated

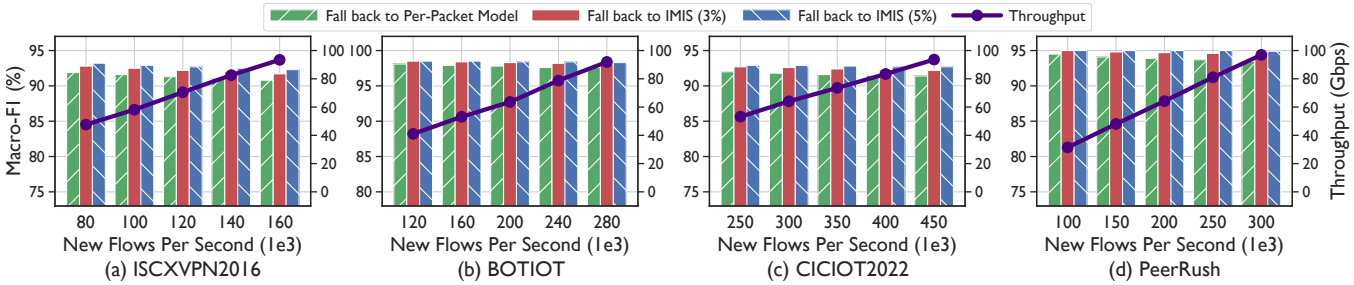


Figure 11: [Testbed] Scaling test of BoS when we progressively increase the aggregate throughput to 100 Gbps.

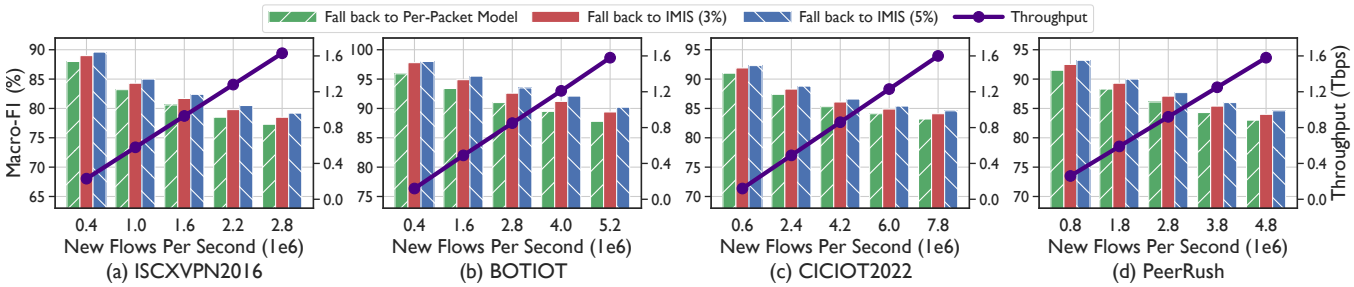


Figure 12: [Simulation] Scaling test of BoS when we progressively increase the aggregate throughput to 1.6 Tbps.

IMIS. When the flow concurrency is high (*i.e.,* Figure 12), this method exhibits reasonable accuracy advantages over falling back to use the per-packet model.

8 Discussion and Related Work

Hardware Dependency. BoS is generic in the sense that all its core designs (*e.g.,* retaining full-precision RNN model weights, using sliding windows to compute unlimited RNN step times) are all realizable using match tables. Since table matching is the universal primitive for any data plane, we expect our designs, with lightweight adaptation, are also deployable on other types of programmable data plane devices.

ML-driven Traffic Analysis. Our community has proposed various ML-powered traffic analysis designs, such as intrusion detection [15, 19, 33], website fingerprinting [11, 41, 46, 65], and encrypted traffic classification [39, 47, 48, 55]. However, it is difficult to directly apply their models in INDP due to the hardware constraints on the data plane.

Advances in the Programmable Data Plane. The flexibility of programmable switches encourages a number of customized applications on the data plane, including network telemetry and monitoring [34, 35, 45], network security [60, 62, 67, 71], and network functions [20]. Additionally, [23, 44] use programmable switches to accelerate ML training, and [25, 53, 70] design auxiliary modules within a switch or leverage off-switch FPGA. Our work focuses on enabling NN-driven INDP using only commodity hardware.

Deployment. BoS is an application-specific system designed for high-throughput and low-latency NN-driven traffic analysis. Therefore, we have not discussed co-deploying other networking functions with BoS on the same programmable switch. Although BoS consumes multiple stages, the SRAM/TCAM consumption per stage is small (see Table 4). Thus, networking functions orthogonal to BoS (*e.g.,* the ECMP in [14]) can be co-deployed with BoS in parallel. Additionally, the latest Tofino chips have almost doubled the number of stages and TCAM/SRAM resources compared to the Tofino 1 chip we use. Thus, we envision that networking functions that may depend on BoS's analysis results (*e.g.,* the traffic policing in [14]) may also be co-deployed with BoS.

9 Conclusion

In this paper, we present BoS, the first INDP design that enables NN-driven traffic analysis at line-speed. The key novelty of BoS is to realize complex RNN computations using a set of novel data plane native operations, and meanwhile to accommodate a transformer-based traffic analysis module via a carefully designed flow escalation mechanism. We implement a prototype of BoS and evaluate it thoroughly on four traffic analysis tasks. The results demonstrate that BoS advances SOTA in both traffic analysis accuracy and scalability.

Acknowledgement

We thank our shepherd Costin Raiciu and the anonymous NSDI reviewers for their insightful feedback. The research is supported in part by the National Key R&D Program of China under Grant 2022YFB2403900, NSFC under Grant 62132011 and Grant 61825204, and Beijing Outstanding Young Scientist Program under Grant BJJWZYJH012019100003011. The corresponding author of this paper is Zhuotao Liu.

References

[1] Soheil Abbasloo, Chen-Yu Yen, and H. Jonathan Chao. Classic Meets Modern: A Pragmatic Learning-Based Congestion Control for the Internet. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2020.

[2] Milad Alizadeh, Javier Fernández-Marqués, Nicholas D. Lane, and Yarin Gal. An Empirical Study of Binary Neural Networks' Optimisation. In *International Conference on Learning Representations (ICLR)*, 2019.

[3] Diogo Barradas, Nuno Santos, and Luís Rodrigues. Effective Detection of Multimedia Protocol Tunneling using Machine Learning. In *USENIX Security Symposium (USENIX Security)*, 2018.

[4] Diogo Barradas, Nuno Santos, Luís Rodrigues, Salvatore Signorello, Fernando MV Ramos, and André Madeira. FlowLens: Enabling Efficient Flow Classification for ML-based Network Security Applications. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 2014.

[6] Broadcom. Trident 5 Programmable Ethernet Switch Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm78800, accessed on Feb. 2024.

[7] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. pForest: In-Network Inference with Random Forests. *arXiv preprint arXiv:1909.05680*, 2019.

[8] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-decoder for Statistical Machine Translation. *arXiv preprint arXiv:1406.1078*, 2014.

[9] CISCO. Global - 2021 Forecast Highlights. https://www.cisco.com/c/dam/m/en_us/solutions/service-provider/vni-forecast-highlights/pdf/Global_2021_Forecast_Highlights.pdf, accessed on Sep. 2023.

[10] Sajjad Dadkhah, Hassan Mahdikhani, Priscilla Kyei Danso, Alireza Zohourian, Kevin Anh Truong, and Ali A. Ghorbani. Towards the Development of a Realistic Multidimensional IoT Profiling Dataset. In *the 19th Annual International Conference on Privacy, Security & Trust (PST)*, 2022.

[11] Xinhao Deng, Qilei Yin, Zhuotao Liu, Xiyuan Zhao, Qi Li, Mingwei Xu, Ke Xu, and Jianping Wu. Robust Multi-tab Website Fingerprinting Attacks in the Wild. In *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[12] Gerard Draper-Gil, Arash Habibi Lashkari, Mohammad Saiful Islam Mamun, and Ali A. Ghorbani. Characterization of Encrypted and VPN Traffic using Time-related Features. In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy (ICISSP)*, 2016.

[13] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[14] Open Networking Foundation. Consolidated switch repo (API, SAI and Nettlink). https://github.com/p4lang/switch/tree/master, accessed on Sep. 2023.

[15] Chuanpu Fu, Qi Li, Meng Shen, and Ke Xu. Realtime Robust Malicious Traffic Detection via Frequency Domain Analysis. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[16] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked Autoencoders are Scalable Vision Learners. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[17] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 1997.

[18] Intel. Data Plane Development Kit. http://www.dpdk.org, accessed on Sep. 2023.

[19] Steve TK Jan, Qingying Hao, Tianrui Hu, Jiameng Pu, Sonal Oswal, Gang Wang, and Bimal Viswanath. Throwing Darts in the Dark? Detecting Bots with Limited Data using Neural Data Augmentation. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[20] Changhun Jung, Sian Kim, Rhongho Jang, David Mohaisen, and DaeHun Nyang. A Scalable and Dynamic ACL System for In-Network Defense. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[21] Piotr Jurkiewicz, Grzegorz Rzym, and Piotr Boryło. Flow Length and Size Distributions in Campus Internet Traffic. *arXiv preprint arXiv:1809.03486*, 2018.

[22] Nickolaos Koroniotis, Nour Moustafa, Elena Sitnikova, and Benjamin Turnbull. Towards the development of realistic botnet dataset in the Internet of Things for network forensic analytics: Bot-IoT dataset. *Future Generation Computer Systems*, 2019.

[23] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network Aggregation for Multi-tenant Learning. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[24] Jong-Hyouk Lee and Kamal Singh. SwitchTree: In-Network Computing and Traffic Analyses with Random Forests. *Neural Computing and Applications*, 2020.

[25] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. Accelerating Distributed Reinforcement Learning with In-Switch Computing. In *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[26] Peng Lin, Kejiang Ye, Yishen Hu, Yanying Lin, and Cheng-Zhong Xu. A Novel Multimodal Deep Learning Framework for Encrypted Traffic Classification. *IEEE/ACM Transactions on Networking (TON)*, 2023.

[27] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollar. Focal Loss for Dense Object Detection. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017.

[28] Xinjie Lin, Gang Xiong, Gaopeng Gou, Zhen Li, Junzheng Shi, and Jing Yu. ET-BERT: A Contextualized Datagram Representation with Pre-training Transformers for Encrypted Traffic Classification. In *Proceedings of the ACM Web Conference (WWW)*, 2022.

[29] Chenyi Liu, Mingwei Xu, Yuan Yang, and Nan Geng. DRL-OR: Deep Reinforcement Learning-based Online Routing for Multi-type Service Requirements. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2021.

[30] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches. In *USENIX Security Symposium (USENIX Security)*, 2021.

[31] Danilo Mandic and Jonathon Chambers. *Recurrent Neural Networks for Prediction: Learning Algorithms, Architectures and Stability*. John Wiley & Sons, Inc., 2001.

[32] Microsoft. Introduction to Receive Side Scaling. https://learn.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling, accessed on Sep. 2023.

[33] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[34] Edgar Costa Molero, Stefano Vissicchio, and Laurent Vanbever. FAst In-Network GraY Failure Detection for ISPs. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2022.

[35] Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. Sketchlib: Enabling Efficient Sketch-based Monitoring on Programmable Switches. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[36] NetFPGA. NetFPGA. https://netfpga.org/, accessed on Sep. 2023.

[37] Netronome. Netronome AgilioTM CX 2x40GbE intelligent server adapter. https://www.netronome.com/media/redactor_files/PB_Agilio_CX_2x40GbE.pdf, accessed on Sep. 2023.

[38] Nvidia. CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit, accessed on Sep. 2023.

[39] Yuqi Qing, Qilei Yin, Xinhao Deng, Yihao Chen, Zhuotao Liu, Kun Sun, Ke Xu, Jia Zhang, and Qi Li. Low-Quality Training Data Only? A Robust Framework for Detecting Encrypted Malicious Network Traffic. In *Network and Distributed System Security Symposium (NDSS)*, 2024.

[40] Babak Rahbarinia, Roberto Perdisci, Andrea Lanzi, and Kang Li. Peerrush: Mining for Unwanted P2P Traffic. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2013.

[41] Vera Rimmer, Davy Preuveneers, Marc Juarez, Tom Van Goethem, and Wouter Joosen. Automated Website Fingerprinting through Deep Learning. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[42] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[43] Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. Can the Network be the AI Accelerator? In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, 2018.

[44] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling Distributed Machine Learning with In-Network Aggregation. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[45] Satadal Sengupta, Hyojoon Kim, and Jennifer Rexford. Continuous In-Network Round-Trip Time Monitoring. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2022.

[46] Meng Shen, Yiting Liu, Liehuang Zhu, Xiaojiang Du, and Jiankun Hu. Fine-grained Webpage Fingerprinting using Only Packet Length Information of Encrypted Traffic. *IEEE Transactions on Information Forensics and Security (TIFS)*, 2020.

[47] Meng Shen, Jinpeng Zhang, Liehuang Zhu, Ke Xu, and Xiaojiang Du. Accurate Decentralized Application Identification via Encrypted Traffic Analysis using Graph Neural Networks. *IEEE Transactions on Information Forensics and Security (TIFS)*, 2021.

[48] Sandra Siby, Marc Juarez, Claudia Diaz, Narseo Vallina-Rodriguez, and Carmela Troncoso. Encrypted DNS–> Privacy? A Traffic Analysis Perspective. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[49] Carlos N. Silla and Alex A. Freitas. A Survey of Hierarchical Classification across Different Application Domains. *Data Mining and Knowledge Discovery*, 2011.

[50] Giuseppe Siracusano and Roberto Bifulco. In-network Neural Networks. *arXiv preprint arXiv:1801.05731*, 2018.

[51] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. Re-architecting Traffic Analysis with Neural Network Interface Cards. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[52] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Hamed Haddadi, Gianni Antichi, and Roberto Bifulco. Running Neural Networks on the NIC. *arXiv preprint arXiv:2009.02353*, 2020.

[53] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. Taurus: A Data Plane Architecture for Per-Packet ML. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.

[54] Mellanox Technologies. BlueField SmartNIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf, accessed on Sep. 2023.

[55] Thijs Van Ede, Riccardo Bortolameotti, Andrea Continella, Jingjing Ren, Daniel J. Dubois, Martina Lindorfer, David Choffnes, Maarten van Steen, and Andreas Peter. Flowprint: Semi-supervised Mobile-app Fingerprinting on Encrypted Network Traffic. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[56] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[57] Bruno Missi Xavier, Rafael Silva Guimarães, Giovanni Comarela, and Magnos Martinello. Programmable Switches for In-networking Classification. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2021.

[58] Guorui Xie, Qing Li, Yutao Dong, Guanglin Duan, Yong Jiang, and Jingpu Duan. Mousika: Enable General In-network Intelligence in Programmable Switches by Knowledge Distillation. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2022.

[59] Renjie Xie, Yixiao Wang, Jiahao Cao, Enhuan Dong, Mingwei Xu, Kun Sun, Qi Li, Licheng Shen, and Menghao Zhang. Rosetta: Enabling Robust TLS Encrypted Traffic Classification in Diverse Network Environments with TCP-Aware Traffic Augmentation. In *USENIX Security Symposium (USENIX Security)*, 2023.

[60] Jiarong Xing, Kuo-Feng Hsu, Yiming Qiu, Ziyang Yang, Hongyi Liu, and Ang Chen. Bedrock: Programmable Network Support for Secure RDMA Systems. In *USENIX Security Symposium (USENIX Security)*, 2022.

[61] Jiarong Xing, Qiao Kang, and Ang Chen. NetWarden: Mitigating Network Covert Channels while Preserving Performance. In *USENIX Security Symposium (USENIX Security)*, 2020.

[62] Jiarong Xing, Wenqing Wu, and Ang Chen. Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries. In *USENIX Security Symposium (USENIX Security)*, 2021.

[63] Siyu Yan, Xiaoliang Wang, Xiaolong Zheng, Yinben Xia, Derui Liu, and Weishan Deng. ACC: Automatic ECN Tuning for High-Speed Datacenter Networks. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2021.

[64] Penghang Yin, Jiancheng Lyu, Shuai Zhang, Stanley J. Osher, Yingyong Qi, and Jack Xin. Understanding Straight-through Estimator in Training Activation Quantized Neural Nets. In *International Conference on Learning Representations (ICLR)*, 2019.

[65] Qilei Yin, Zhuotao Liu, Qi Li, Tao Wang, Qian Wang, Chao Shen, and Yixiao Xu. An Automated Multi-Tab Website Fingerprinting Attack. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2021.

[66] Ruijie Zhao, Mingwei Zhan, Xianwen Deng, Yanhao Wang, Yijun Wang, Guan Gui, and Zhi Xue. Yet another Traffic Classifier: A Masked Autoencoder based Traffic Transformer with Multi-Level Flow Representation. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2023.

[67] Ziming Zhao, Zhuotao Liu, Huan Chen, Fan Zhang, Zhuoxue Song, and Zhaoxuan Li. Effective DDoS Mitigation via ML-Driven In-network Traffic Shaping. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2024.

[68] Changgang Zheng, Zhaoqi Xiong, Thanh T. Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. IIsy: Practical In-Network Classification. *arXiv preprint arXiv:2205.08243*, 2022.

[69] Changgang Zheng and Noa Zilberman. Planter: Seeding Trees within Switches. In *Proceedings of the SIGCOMM'21 Poster and Demo Sessions*, 2021.

[70] Zhizhen Zhong, Weiyang Wang, Manya Ghobadi, Alexander Sludds, Ryan Hamerly, Liane Bernstein, and Dirk Englund. IOI: In-network Optical Inference. In *Proceedings of the ACM SIGCOMM 2021 Workshop on Optical Systems*, 2021.

[71] Guangmeng Zhou, Zhuotao Liu, Chuanpu Fu, Qi Li, and Ke Xu. An Efficient Design of Intelligent Network Data Plane. In *USENIX Security Symposium (USENIX Security)*, 2023.

[72] Guihua Zhou, Guo Chen, Fusheng Lin, Tingting Xu, Dehui Wei, Jianbing Wu, Li Chen, Yuanwei Lu, Andrew Qu, Hua Shao, and Hongbo Jiang. Primus: Fast and Robust Centralized Routing for Large-scale Data Center Networks. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2021.

A Appendix

A.1 Model Realization on the Data Plane

In this segment, we supplement additional design details regarding deploying our binary RNN on the programmable data plane.

A.1.1 Intuitive Number Comparison

The foundation to realize argmax is number comparison. There are two types of approaches to compare numbers on the programmable data plane. The first is based on conditional statements. Specifically, a simple statement to compare num_A and num_B is

```
if (num_A > num_B) act_A; else act_B;
```

which, unfortunately, is not compiled because the condition is too complex. We could avoid complex condition by the following statement

```
tmp = num_A - num_B;
if (tmp > 0) act_A; else act_B;
```

which takes at least two switch stages. More crucially, it is difficult to scale statement-based approaches to compare n numbers since it would take $n \times (n-1)/2$ differences.

The alternative approach is designing a match-action table which takes the num_A and num_B as keys (*e.g.,* the concatenation of their bits) and performs act_A or act_B according to the lookup result. Meanwhile, we can easily extend width of keys to realize the comparison of multiple numbers, which is exactly the argmax operation. Yet, the drawback is the explosion of required table entries: to obtain the maximum number among n values, each with m-bits, it takes 2^{nm} entries to enumerate all possible key-value combinations.

A.1.2 Ternary-Matching Based Design

In § 5.2, we discuss the implementation of argmax operation with ternary matching. We define the problem as following conditions and restrictions.

1. There are n numbers of m bit(s) each, where $n, m \geq 1$.
2. There is a predefined order to determine which number to select when there is a tie for maximum value.
3. Only one ternary matching is allowed, i.e., the calculation before matching is prohibited.

We denote the number of table entries as $F(n,m)$. Based on the basic optimization in § 5.2, we get the following recurrence relationship of $F(n,m)$.

$$F(n,m) = 2 \times F(n,m-1) + \sum_{i=1}^{n-1} \binom{n}{i} F(i,m-1), n \geq 2, m \geq 2 \tag{2}$$

We explain the meaning of the above equation as follows. The entries should cover all possible combinations of the n numbers. We consider all the combinations in different categories classified according to the combination of most significant bits (MSBs). Among all 2^n categories, there are $\binom{n}{i}$ categories where i numbers are with $\mathsf{MSB} = 1$ and $n-i$ numbers are with $\mathsf{MSB} = 0$ ($i \in [1, n-1]$). In these categories, we do not further consider those $n-i$ numbers with $\mathsf{MSB} = 0$, and only focus on the possible combinations of the i numbers with $\mathsf{MSB} = 1$, which are sub-problems with $n' = i, m' = m-1$ and require $\binom{n}{i} F(i, m-1)$ entries in total. In the other two categories (all $\mathsf{MSB} = 0$ or all $\mathsf{MSB} = 1$), we continue to focus on the possible combinations of all the n numbers, which are both sub-problems with $n' = n, m' = m-1$ and require $2 \times F(n, m-1)$ entries in total.

After the two optimizations described in § 5.2, the recurrence relation of $F(n,m)$ is given as follows.

$$F(n,m) = F(n,m-1) + \sum_{i=1}^{n-1} \binom{n}{i} F(i,m-1), n \geq 2, m \geq 2 \tag{3}$$

$$F(n,1) = n, n \geq 1 \tag{4}$$

$$F(1,m) = 1, m \geq 1 \tag{5}$$

By solving this iterative formula, we obtain $F(n,m) = nm^{n-1}$. We provide the derivation process from Equation (3) as follows. First, let $F(0,m) = 0, m \geq 1$, which is consistent with the Equation (4). Then the Equation (3) can be written as:

$$F(n,m) = \sum_{i=0}^{n} \binom{n}{i} F(i,m-1), n \geq 0, m \geq 2 \tag{6}$$

Then, we transform the formula and get:

$$F(n,m) = \sum_{i=0}^{n} \frac{n!}{i!(n-i)!} F(i,m-1), n \geq 0, m \geq 2 \tag{7}$$

$$\frac{F(n,m)}{n!} = \sum_{i=0}^{n} \frac{F(i,m-1)}{i!} \frac{1}{(n-i)!}, n \geq 0, m \geq 2 \tag{8}$$

We denote $\frac{F(n,m)}{n!}$ as ${}^m g_n$, and $\frac{1}{n!}$ as h_n. Then we construct the generating function of ${}^m g$ and h.

$${}^m G(x) = \sum_{n=0} {}^m g_n x^n = \sum_{n=0} \frac{F(n,m)}{n!} x^n, m \geq 1 \tag{9}$$

$$H(x) = \sum_{n=0} h_n x^n = \sum_{n=0} \frac{x^n}{n!} = e^x \tag{10}$$

Table 5: The no. of entries required for different m, n.

No. of Entries	Opt 1 & 2	Opt 2 only	Opt 1 only	Base Design	2^{mn}
n=3,m=16	768	2949123	863	4587523	2.81e14
n=4,m=8	2048	44028	2788	76028	4.29e9
n=5,m=5	3125	10245	5472	21077	3.36e7
n=6,m=4	6144	10890	13438	26978	1.68e7

We can obtain the recursive relations between ${}^{m}G$ and ${}^{m-1}G$ from Equation (8).

$$ {}^{m}G =^{m-1} G \times H, m \geq 2 \tag{11} $$

And for $m = 1$, we have

$$ \begin{aligned} {}^{1}G(x) &= \sum_{n=0} {}^{1}g_n x^n = \sum_{n=0} \frac{F(n,1)}{n!} x^n \\ &= \sum_{n=0} \frac{n}{n!} x^n = \sum_{n=1} \frac{1}{(n-1)!} x^n \\ &= x \sum_{n=0} \frac{1}{n!} x^n = x e^x \end{aligned} \tag{12} $$

With the mathematical induction, we can get

$$ {}^{m}G(x) = x e^{mx}, m \geq 1 \tag{13} $$

Compared with the Equation (9) and we can get

$$ \begin{aligned} F(n,m) &=^{m} G^{(n)}(0) = (x e^{mx})^{(n)} \Big|_{x=0} \\ &= (nm^{n-1} + m^n x) e^{mx} \Big|_{x=0} \\ &= nm^{n-1}, n \geq 1, m \geq 1 \end{aligned} \tag{14} $$

We can verify the result using mathematical induction.

$$ \begin{aligned} & F(n,m-1) + \sum_{i=1}^{n-1} \binom{n}{i} F(i,m-1) \\ &= \sum_{i=1}^{n} \binom{n}{i} i(m-1)^{i-1} \\ &= n \sum_{i=1}^{n} \binom{n-1}{i-1} (m-1)^{i-1} \\ &= n \sum_{i=0}^{n-1} \binom{n-1}{i} (m-1)^{i} \\ &= nm^{n-1} = F(n,m) \end{aligned} \tag{15} $$

In Table 5, we list the number of entries required for different combinations of m and n. The results demonstrate that our design, augmented by two optimizations, significantly reduces table consumption for achieving the argmax operation.

A.1.3 Packet Counters

Because the number of packets in a flow is unknown in advance, statically allocating a fixed width of bits for packet counters may result in buffer overflow. Meanwhile, as described in § 5.1, we need to perform modulo operations on packet count (*i.e.,* pktcnt % $(S-1)$) when storing embedding vectors. Thus, packet counting in BoS is designed based on two parallel counters: the first counter increases from 1, and stops at S (the sliding window size). For the i^{th} packet, it returns i if $i < S$, otherwise it returns S. The second counter increases from 0 and cycles back to 0 after $S-2$, simulating the modulo operation. Thus, when the number of arrived packets in the flow exceeds S, the first counter essentially becomes a flag indicating that index for the ring buffer (storing embedding vectors) can be read from the second counter.

A.1.4 Flow Management

BoS relies on stateful storage to maintain per-flow state. Prior art [4, 61] relies on the control plane to allocate non-conflicting storage indices for different flows. To achieve line-speed traffic analysis, BoS relies on the readily available hardware hashing to allocate flow storage indices. In particular, the storage index for flow f is computed as $\mathcal{H}(f(\text{5-tuple}) \% \text{N})$, where $\mathcal{H}$ is the hash function, and N represents the total number of continuous per-flow storage blocks allocated for maintaining per-flow state.

Both hash and modulo operations may result in flow index collisions, *i.e.,* two different flows (with different 5-tuples) may receive the same storage index. To avoid confusions, BoS stores a tuple {TrueID, timestamp} alongside the storage index, where TrueID represents the actual flow identifier[2] and timestamp represents the latest packet arrival time for the flow. When storage indices collide, BoS allows the new flow to take the occupied storage only if the existing flow is timed out (*i.e.,* the stored timestamp is earlier than a predefined threshold). Otherwise, the new arrived flow falls back to use the per-packet tree model trained using only per-packet features, or falls back to IMIS; see discussions in § 7.3.

When developing the prototype of BoS, we observe a possible corner case for flow management. Specifically, the switch has multiple forwarding pipes, each of which has several processing stages. To support more complex RNN models, we can simultaneously use the stages in both the ingress and egress pipes. However, if multiple ingress pipes could be mapped to the same egress pipe (*e.g.,* traffic entered from both pipe A and pipe B may exit from pipe A), we would need to deploy a flow management module in both the ingress and egress pipe, because flows that do not collide in their ingress pipes may collide in the egress pipe. We have not encountered this corner case even in our scaling experiments (see § 7.3), and therefore we only deploy the flow management module in the ingress pipe.

[2] To avoid resubmitting or recirculating packets, the read and write of the tuple need to be finished in an atomic operation. This restricts the length of TrueID so that we cannot directly use 5 tuple as the TrueID. Thus, we leverage a different hash function H' to calculate the TrueID as $H'(f(\text{5-tuple}))$.

A.1.5 Per-packet Fallback Model

When the flow manager cannot allocate storage for a new flow, BoS falls back to analyzing the packets of that flow using a tree model trained only using per-packet features. Specifically, we use a 2×9 Random Forest model (2 trees with max depth 9), and use the same per-packet features as in [71] (*e.g.,* packet length, TTL, Type of Service, TCP offset). We apply the coding mechanism from NetBeacon [71] to deploy this tree model on the data plane alongside our binary RNN model.

A.1.6 The Pre-analysis Issue

As discussed in § 4.3, we employ a sliding window mechanism in our binary RNN inference where the model continuously processes packet segments. The length of the segment is a hyper-parameter S (set to 8 in our prototype). As a result, the very first $S-1$ packets of a flow cannot form a complete segment. This results in the pre-analysis issue: the inference results on these packets may be inaccurate because the model simply has not observed enough information. Any model-driven (or data-driven) traffic analysis approach has this limitation.

To avoid premature inference results caused by the pre-analysis problem, BoS regards the first $S-1$ packets of a flow as *pre-analysis packets*, and only starts to produce inference results for the S^{th} and subsequent packets (*i.e.,* any inference result output by BoS is based on at least one full segment). The protocol for forwarding these pre-analysis packets should be application-specific. For instance, in security-oriented task, BoS can forward pre-analysis packets via a dedicated low priority queue so that a strategic adversary cannot overwhelm the network by sending very short flows (less than S packets). In other tasks (*e.g.,* an inbound gateway on a campus/enterprise that loads balance different types of traffic received from the Internet), simply forwarding these pre-analysis packets may be sufficient, considering the average length of campus Internet flows ($\sim$120) [21] is much larger than S (8 in our prototype). Finally, it is possible to employ another learning model trained only on per-packet features (such as [58]) to process these pre-analysis packets.

A.2 Prototype Implementation

In this segment, we supplement additional details about our implementation.

A.2.1 On-Switch RNN Inference

Component Overview. We plot the hardware implementation of the binary RNN in BoS in Figure 8. The top part shows the simplified dependency graph of all components. We plots two types of dependency: if a has data dependency on b, then the input data of a is (partially) provided by b; if a has control dependency on b, then the execution of a is determined by b. For instance, the GRU tables have data dependency on embedding vector storage; the window CTR (counting the number of windows/segments) has control dependency on the PKT CTR-1 (indicating whether the number of received packet is no less than S).

Per-Stage Breakdown. The bottom-right part shows the breakdown of stage usage for deploying a BoS model with the hyper-parameters shown in the bottom-left part. We use the stages in both ingress and egress pipeline. The k^{th} ingress stage and k^{th} egress stage share the same underlying hardware resource.

We first introduce the stage usage in ingress. In the stage 0, besides calculating the flow index and TrueID for flow management, it also executes embedding of packet length, since it has no other dependency. Then, the FlowInfo tuple (*i.e.,* $\{\text{TrueID}, \text{timestamp}\}$) is stored in stage 1 for flow collision avoidance. Flow management is only necessary in the ingress pipeline (see § A.1.4). The inter-packet delay (IPD) embedding is implemented using the following three stages: stage 2 stores the last packet timestamp, stage 3 obtains IPD by subtracting the current packet arrival time with the last timestamp, and stage 4 computes the embedding of IPD. In stage 5, an FC layer takes in the packet length embedding and IPD embedding to output an embedding vector, which is stored using 7 bins and dispatched to corresponding GRU tables in stages 6 to 8. All seven bins cannot be allocated into one stage because only 4 registers (register arrays) are allowed in one stage. In the last three stages of ingress, the first four GRU tables (*i.e.,* $\text{GRU}_{1,2,3,4}$) are placed sequentially. The first two GRU tables (*i.e.,* $\text{GRU}_{1,2}$) are implemented with one match-action table, *i.e.,* $h{=}\text{GRU}_2(\text{GRU}_1(0, ev_1), ev_2)$ is merged as $h{=}(\text{GRU}_2 \circ \text{GRU}_1)(0, ev_1, ev_2)$.

In the egress pipeline, the remaining four GRU tables (*i.e.,* $\text{GRU}_{5,6,7,8}$) are placed from stage 0 to stage 3. The output layer is merged with GRU_8, *i.e.,* $C{=}\text{Output}(\text{GRU}_8(h, ev_8))$ is merged as $C{=}(\text{Output} \circ \text{GRU}_8)(h, ev_8)$. The counters to accumulate per-class probabilities ($\text{CPR}_{1..6}$) are spread in stage 4 and 5. To accumulate the probability vectors (*i.e.,* the intermediate results) on the data plane, we quantize the probability for a class to an integer from 0 to 15. Considering the reset period of 128 packets, the width of cumulative probability is $\lceil log_2(16\times128)\rceil{=}11$. To implement argmax for $n{=}6, m{=}11$, we split it into three sequential argmax operations, two for $n{=}3, m{=}11$ and one for $n{=}2, m{=}11$, *i.e.,* comparing the first three numbers in stage 5, then comparing the rest three numbers in stage 6, then comparing the two winners in stage 7. Finally, the escalation logic is implemented in stage 8 to 9. To obtain the classification confidence for a packet, we do not actually divide the largest accumulative probability with wincnt (the total number of intermediate results for the flow), as division is not supported on the data plane. Instead, we compare the probability with $\mathbb{T}_{\text{conf}} \times \text{wincnt}$, which is divided into a subtraction and a comparison with 0. The subtraction

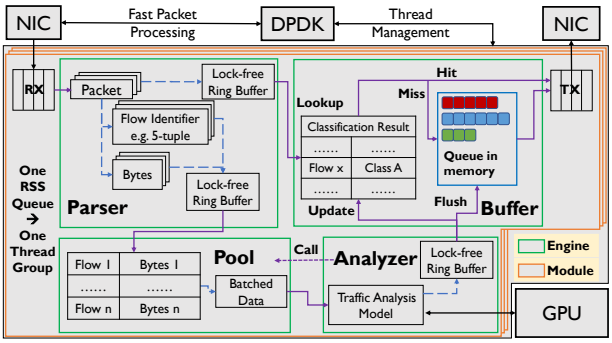


Figure 13: The architecture of IMIS.

is performed in the *action* of the winning case of argmax match-action table. And the comparison with 0 is executed while reading/updating the counter that stores the number of ambiguous packets in stage 8. If the counter exceeds T_{esc}, the packet is escalated to IMIS.

Escalation Flag. Due to the limited number of stages in the ingress pipeline of the Tofino 1 switch, we must use both the ingress and egress pipelines in our prototype. However, this poses a challenge because the egress port of a packet must be determined in the ingress pipeline, but we cannot determine whether a packet should be escalated to IMIS until all operations in the egress pipeline have completed. To address this challenge, we store an escalation flag in the ingress pipeline so that the egress port for a packet can be properly determined. We update the escalation flag through egress-to-egress mirroring and recirculating.

A.2.2 Implementation of IMIS

The architecture of IMIS is plotted in Figure 13. We use Intel Data Plane Development Kit (DPDK version 20.11.9 LTS) [18] to enable multiple NIC RX/TX queues, each of which is bound to one analysis module. The Receive Side Scaling (RSS) [32] is enabled to efficiently distribute traffic to each analysis module.

Analysis Module. The parser engine uses DPDK APIs to parse flow identifier (*e.g.,* 5-tuple) and the raw bytes from the input traffic. It stores the parsing results into a lock-free ring buffer consumed by the pool engine to maintain per-flow state and perform batch arrangement. After obtaining the parsing result of a packet, the packet is sent to another lock-free ring buffer consumed by the buffer engine to perform egress queuing according to model inference results. As our transformer-based model only uses the first 5 packets in a flow for inference, the subsequent packets sent by the flow will be forwarded to the buffer engine directly without raw bytes extraction.

The pool engine translates the streamed parsing results into batched data to facilitate model inference. Specifically, it continuously fetches the raw byte features of packets from the lock-free ring buffer linked to the parser engine, and organizes them as per-flow state. When it receives a call from the analyzer engine, the pool engine selects flows according to their timestamps to form a batch of inputs, and sends the batch to the analyzer engine for inference. If a selected flow has fewer than 5 packets, the pool engine pads its data with zeros. The inference result obtained for this flow is considered to intermediate, and the pool engine may select this flow again in the next round.

To accelerate model inference, the analyzer engine uses CUDA (version 11.7) [38] to interact with the auxiliary GPU card. Specifically, it continuously requests input batches from the pool engine. Upon receiving a batch, the analyzer engine executes inference on the GPU and sends the results to a lock-free ring buffer consumed by the buffer engine.

The buffer engine continuously fetches the latest inference results from the analyzer engine and uses the results to release packets. Upon receiving a packet from the parser engine, the buffer engine checks if the inference result for the packet's flow has been determined. If so, the packet is released immediately. Otherwise, the packet is placed in the egress queue for its flow to wait for the inference result. When the buffer engine receives a flow inference result from the analyzer engine, it releases all packets in the egress queue for that flow.

The buffer engine keeps fetching the latest inference results from the analyzer engine, and uses the results to release packets. Upon receiving a packet from the parser engine, the buffer engine checks if the inference result for the packet's flow has been determined. If so, the packet is released immediately. Otherwise, the packet is placed in the egress queue for its flow to wait for the inference result. When the buffer engine receives a flow inference result from the analyzer engine, it releases all packets in the egress queue for that flow.

A.3 Additional Details about Testbed

We use a Wedge 100BF-32X programmable switch with 2 pipes and 32×100 Gbps ports to deploy the on-switch RNN in BoS. The version of SDE is 9.7.0. The off-switch IMIS is hosted on server with two Intel(R) Xeon(R) Gold 6348 CPUs (2×28 cores), Ubuntu 20.04.1, 512 GB memory, and one Mellanox 100 Gbps NIC with two ports that support DPDK (version 20.11). We reserve 160 GB memory as huge pages for DPDK (80 GB/NUMA Node), and an NVIDIA A100 GPU is attached to IMIS. All physical cores for parser engines, pool engines and buffer engines are on the same Node with NIC, and all the physical cores for analyzer engines are on the same Node with an auxiliary GPU.

Scaling the On-switch Analysis Beyond One Pipe. The on-switch analysis of our current prototype is implemented using one switch pipe. The complexity of scaling the analysis beyond one pipe depends on whether cross-pipe traffic forwarding is allowed. Specifically, if the traffic forwarding for each pipe is self-contained (*i.e.,* the traffic ingressing from one pipe will only exit from this pipe), we can easily operate

multiple pipes independently, where each pipe hosts an instance of our on-switch RNN and processes traffic in parallel. However, when the flows entering from different pipes can eventually exit via the same pipe, we have to deploy our flow management modules in both the ingress and egress pipes (as we have discussed in § A.1.4). In this case, the flows allocated dedicated per-flow storage within their ingress pipes may still end up using the per-packet model if their storage indices collide when exiting from the same egress pipe. This would lead to the underutilization of storage resources initially reserved for these flows within their ingress pipes.

Runtime Programmability. The on-switch analysis model of BoS can be programmed in runtime. Specifically, the weights of RNN layers, the escalation thresholds, the number of classification classes, and widths of the inputs and outputs of each layer (*i.e.,* the number of binary neurons) are all programmable via the control plane. For instance, the weights can be reconfigured by updating the table entries from the control plane.

On-switch Statistics Collection. To collect the evaluation results from our testbed, we use the second pipe on our switch to implement a result collection module. Specifically, we allocate registers to count the numbers of escalated packets, packets analyzed by per-packet model, packets analyzed by binary RNN, and pre-analysis packets. Further, we allocate a register array for reporting the on-switch analysis precision and recall for each class, using the combination of ground-truth label and predict label as index. We read these registers from the control plane to obtain the raw data for calculating the macro-F1 scores.

Flow Replayer. To generate traffic according to our pcap files, we investigate both tcprelay and DPDK pktgen. We choose to use pktgen because it can generate high-throughput traffic that saturates the physical 100 Gbps NIC on our testbed. Yet, the key problem of pktgen is that it fails to honor the packet timestamps when sending traffic. However, the on-switch RNN relies on inter-packet delays for inference. To work around this issue, we embed the desired timestamp of each packet within the MAC address field of its Ethernet frame. The on-switch analysis pipeline reads this field for flow management and inference. We create 32 pcap files throughout the evaluation. When the flow replayer sends an excessively large pcap file that cannot be loaded into the memory at once, it breaks the file into smaller slices and replays these slices sequentially.

Stress Test of Standalone IMIS. To stress test the system performance of IMIS (§ 7.3), we generate flows on a server with DPDK packet generator (pktgen version 23.06). These flows are sent directly to the server where we deploy IMIS, bypassing the on-switch analysis. To generate a burst of concurrent flows, the packet generator repeatedly sends packets within a group of selected 5-tuples and the packet size is fixed as 512 bytes.

A.4 Additional Details about Datasets

Data Pre-processing. For every dataset used in our evaluations, we collect flow records from the original pcap files using the following procedure. (i) We collect the original pcap files for each class in the dataset separately, and all the flow records extracted from a pcap file are labelled as the class of this file. (ii) For each pcap file, we collect the TCP and UDP packets of IPv4, and remove other irrelevant packets, *e.g.,* packets of Domain Name System (DNS), Address Resolution Protocol (ARP), Dynamic Host Configuration Protocol (DHCP) and so on. (iii) We split a clean pcap file by five tuple, and further split packets of the same five tuple into flow records by inter-packet delays. Specifically, if the inter-packet delay between two packets is greater than 256 ms, we consider the latter packet as the first packet of a new flow record. This is consistent with our online inference where we consider a flow is completed if we do not receive new packets for the flow for 256 ms. (iv) 80% of flow records in a dataset are used as the training set and the remaining records are used as testing set.

Traffic Analysis Tasks. We evaluate BoS using the following tasks.

- Encrypted traffic classification on VPN. This task classifies traffic encrypted by Virtual Private Networks (VPNs). We use the ISCXVPN2016 [12] dataset, which contains 7 categories of communication applications captured through the Canadian Institute for Cybersecurity in both VPN and non-VPN. We process the original pcap files for 6 classes of VPN flows, including Email, Chat, Streaming, FTP, VoIP, and P2P. We exclude the Browsing class in our evaluation because some of the applications used for generating Email, Streaming, VoIP packets are web-based, resulting in significant noises, as explained in [12]. The number of flows in each of the six classes is 613, 2350, 375, 1789, 3495, and 1130, respectively.
- Botnet traffic classification on IoT. This task classifies different botnet traffic collected from the Internet of Things (IoT) systems. We process the original pcap files for 4 classes of flows (Data Exfiltration, Key Logging, OS Scan, Service Scan) from the BOTIOT [22] dataset, collected in a realistic network environment deployed in the Cyber Range Lab of UNSW Canberra. The number of flows in each class is 353, 427, 1593, and 7423, respectively.
- Behavioral analysis of IoT Devices. This task classifies traffic generated by IoT devices in different working states. We collect the original pcap files for 3 classes (Power, Idle, Interact) from the CICIOT2022 [10] dataset, which contains 40 devices of audio, camera, home automation and so on. We process the original pcap files for the Power and Interact classes, and select one day from the 30 days of Idle pcap

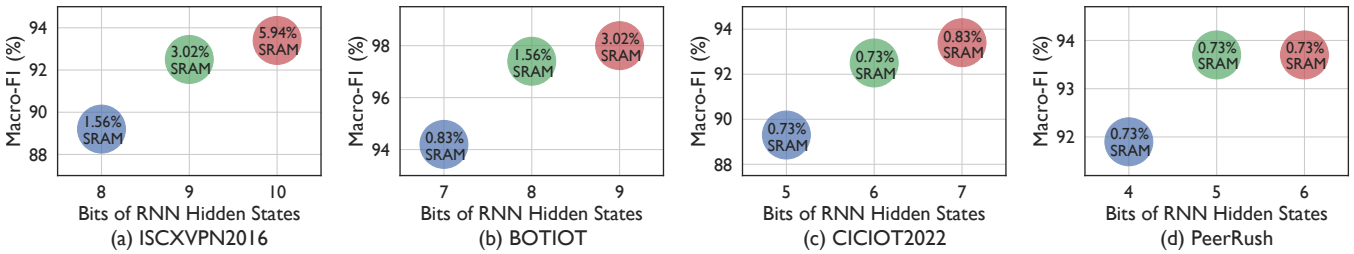


Figure 14: [Testbed] The traffic analysis accuracy given different binary RNN model sizes.

files. The number of flows in each class is 1131, 4382, and 1154, respectively.

- P2P application fingerprinting. This task classifies P2P application traffic. We process the original pcap files for 3 classes (eMule, uTorrent, and Vuze) from the PeerRush dataset [40]. Each class captures one hour of traffic. The number of flows in each class is 20919, 9499, and 7846, respectively.

A.5 Reproducing [71] and [51]

We reproduce two recent art NetBeacon [71] and N3IC [51] for evaluation.

- NetBeacon [71]: a reproduced version of NetBeacon, which deploys multi-phase tree-based models on switch using both flow-level features and per-packet features. We use the same per-packet features as in [71], and use the max, min, mean, and variance of the packet size and IPD as flow-level features. The inference points are located at the $\{8^{\text{th}}, 32^{\text{nd}}, 256^{\text{th}}, 512^{\text{nd}}, 2048^{\text{th}}\}$ packet. For each phase we train a 3×7 (3 trees with max depth 7) Random Forest model (their largest model).
- N3IC [51]: a reproduced version of N3IC, which deploys binary MLP on SmartNIC using both statistical flow-level features and per-packet features. We use the same features and phases as NetBeacon for fair comparison, and for each phase the number of neurons in the hidden layers is [128, 64, 10] (their largest model). Note that N3IC deploys binary MLP on SmartNIC but the model cannot be deployed on P4 switches due to hardware resource constraints. Thus, we simulate the switch-side traffic management logic and the binary MLP inference in software to to obtain the traffic analysis results for N3IC.

A.6 Binary RNN Model Complexity

The number of bits allocated to store the RNN hidden states determines both the performance of our RNN model and the size of match-action table for a GRU layer. In Figure 14, we present the performance of BoS under different bit lengths. The default bit lengths used in four tasks are 9, 8, 6, and 5, respectively. This is because further increasing the bit lengths does not significantly improve F1 scores, while it does increase the SRAM consumption (especially for the first two tasks). When the bit length is smaller than 6, the size of one GRU table is smaller than the minimum allocation unit of SRAM. Thus, further reducing bit lengths will not reduce SRAM consumption.