An FPGA Integrated Programmable Switch Architecture for Data Plane DNN Inference

Xiangyu Gao, *Graduate Student Member, IEEE*, Tong Li, *Member, IEEE*, Yinchao Zhang, *Member, IEEE*, Ziqiang Wang, Xiangsheng Zeng, Liang Zhu, Su Yao, *Senior Member, IEEE*, and Ke Xu, *Fellow, IEEE*

***Abstract*—Machine learning (ML) is increasingly used in network data planes for advanced traffic analysis, but existing solutions (such as FlowLens, N3IC, BoS) still struggle to simultaneously achieve low latency, high throughput, and high accuracy. To address these challenges, we present FENIX, a hybrid in-network ML system that performs feature extraction on programmable switch ASICs and deep neural network inference on FPGAs. FENIX introduces a Data Engine that leverages a probabilistic token bucket algorithm to control the sending rate of feature streams, effectively addressing the throughput gap between programmable switch ASICs and FPGAs. In addition, FENIX designs a Model Engine to enable high-accuracy deep neural network inference in the network, overcoming the difficulty of deploying complex models on resource-constrained switch chips. We implement FENIX on a programmable switch platform that integrates a Tofino ASIC and a ZU19EG FPGA directly, and evaluate it on real-world network traffic datasets. Our results show that FENIX achieves microsecond-level inference latency and multi-terabit throughput with low hardware overhead, and delivers over 90% accuracy on mainstream network traffic classification tasks, outperforming the state of the art.**

***Index Terms*—Intelligent dataplane, programmable switch, FPGA-enhanced architecture.**

Received 6 January 2026; revised 23 June 2026; accepted 14 August 2026; approved by IEEE TRANSACTIONS ON NETWORKING Editor Y. Lee. Date of publication 20 August 2026; date of current version 4 September 2026. This work was supported in part by the National Science Fund for Distinguished Young Scholars under Grant 62425201; in part by the National Natural Science Foundation of China Innovative Research Group Project under Grant 62221003; in part by the National Natural Science Foundation of China under Grant 62132011, Grant 62472240, Grant 62394322, Grant U22B2031, and Grant 62572473; in part by the Beijing Natural Science Foundation under Grant L2603007; in part by the Taishan Scholar Foundation of Shandong Province under Grant tstp20250724; and in part by the State Key Laboratory of Internet Architecture under Grant HLW2025ZD17. *(Corresponding authors: Ke Xu; Su Yao.)*

Xiangyu Gao is with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, and also with the State Key Laboratory of Internet Architecture, Beijing 100084, China (e-mail: gao-xy24@mails.tsinghua.edu.cn).

Tong Li is with Renmin University of China, Beijing 100872, China, and also with the State Key Laboratory of Internet Architecture, Beijing 100084, China (e-mail: tong.li@ruc.edu.cn).

Yinchao Zhang is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China, and also with the State Key Laboratory of Internet Architecture, Beijing 100084, China (e-mail: afireswallow@gmail.com).

Ziqiang Wang is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: ziqiangwang9@gmail.com).

Xiangsheng Zeng is with Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: xiangshengz@hust.edu.cn).

Liang Zhu is with Zhongguancun Laboratory, Beijing 100094, China (e-mail: zhuliang-x@hotmail.com).

Su Yao is with Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, Beijing 100084, China, and also with the State Key Laboratory of Internet Architecture, Beijing 100084, China (e-mail: yaosu@tsinghua.edu.cn).

Ke Xu is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China, and also with Zhongguancun Laboratory, Beijing 100094, China (e-mail: xuke@tsinghua.edu.cn).

Digital Object Identifier 10.1109/TON.2026.3725793

I. INTRODUCTION

MACHINE learning (ML) is increasingly used for in-network traffic analysis. Recent work demonstrates that both traditional ML techniques, including decision trees, random forests, and support vector machines, and deep learning (DL) approaches can significantly improve networking tasks, including malicious traffic detection [1], [2], [3], [4] and application traffic classification [3], [5], [6]. These advances have been enabled by the emergence of programmable network devices, including P4 switches [7], NetFPGA [8], and SmartNICs [9]. By integrating data-driven learning models with traditional rule-based mechanisms, these systems enable more adaptive and responsive in-network traffic analysis. However, deploying these models inside the network data path creates a three-way tension among inference latency, forwarding throughput, and model complexity.

Early intelligent network designs typically split responsibilities between planes: the programmable data plane extracts traffic features, while the control plane performs ML inference. Although this leverages existing programmable infrastructure, it introduces communication latencies, often milliseconds [5], [10]. Such delays are especially problematic for time-sensitive applications like intrusion detection, as analysis may lag behind real-time traffic (see § VII for details).

To address these latency challenges, researchers have explored moving ML inference closer to the data path. One promising approach is implementing inference directly on programmable devices such as SmartNICs, as demonstrated by systems like N3IC [11]. This offloads computation from endpoint CPUs and significantly reduces inference latency. However, new bottlenecks arise: for example, N3IC achieves maximum throughput of only 40 Gbps, and even the latest commercial SmartNICs are limited to 400 Gbps [12]. Such throughput is often insufficient for core networks, where switch ASICs handle 4 to 12 Tbps [13], [14], far exceeding current SmartNIC capabilities. While deploying multiple NICs can scale throughput, this approach introduces significant costs and new scheduling challenges, as packet scheduling consumes significant CPU resources [15].

2998-4157 © 2026 IEEE. All rights reserved, including rights for text and data mining, and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:09:32 UTC from IEEE Xplore. Restrictions apply.

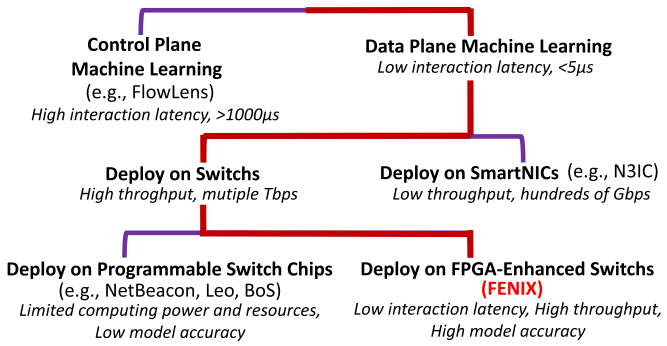


Fig. 1. Design space in intelligent network.

Recent advances in in-network ML have led researchers to implement ML models directly on programmable switch ASICs. For example, systems such as Leo [4] and NetBeacon [3] deploy decision tree and random forest models on Tofino switches using Match Action Tables (MAT), while BoS [6] demonstrates the feasibility of running binary recurrent neural networks (RNNs) on similar hardware. These approaches can potentially enable line-rate ML inference at significantly higher throughput. However, they are typically subject to stringent computational resource constraints, which limit their effectiveness for complex multi-classification tasks (see § VII for details). Architecture-oriented systems such as Taurus [10] further show that specialized compute blocks can extend PISA-like pipelines for per-packet ML. Taurus validates this direction through FPGA prototyping and ASIC synthesis, but it leaves open a different deployment question: how to combine a commercial programmable switch with an attached FPGA while keeping the switch pipeline nonblocking under high-rate traffic.

The remaining gap is a deployable architecture that preserves commercial switch line-rate processing, uses reconfigurable hardware for DNN inference, and controls the bandwidth mismatch between the two chips.In this paper, we present FENIX (FPGA-Enabled Neural Inference eXecution for network switches), a system designed to achieve low latency, high throughput, and high accuracy for in-network intelligent traffic analysis.

Figure 1 positions FENIX in this design space. Compared with control-plane inference, FENIX avoids millisecond-scale switch-controller interactions. Compared with SmartNIC-based inference, it keeps feature extraction on a multi-terabit switch ASIC. Compared with switch-only ML designs, it offloads arithmetic-heavy DNN execution to an attached FPGA while using rate control to avoid stalling the data plane.

FENIX consists of two main components: the Data Engine and Model Engine, which enable in-network traffic analysis on the data plane. The Data Engine extracts features from high-throughput traffic, while the Model Engine performs ML inference on these features. This design enables FENIX to achieve three key goals: *low latency* by avoiding software processing, *high throughput* through efficient feature extraction, and *high accuracy* by supporting full DNN models with minimal quantization loss.

Contributions. The main contribution of this paper is the design, implementation, and evaluation of FENIX, a deployable commercial switch–FPGA system for data-plane DNN inference. FENIX keeps line-rate parsing, feature extraction, and flow-state updates on a Tofino switch, offloads DNN execution to a ZU19EG FPGA, and uses probabilistic rate limiting to make the offload stable under high-rate traffic.Over six months, we designed and manufactured a custom switch, integrating a programmable Tofino switch chip and a ZU19EG FPGA on Printed Circuit Board (PCB). We evaluate FENIX on challenging tasks, including VPN encrypted traffic classification and malware detection. Experimental results show that FENIX achieves up to 537× lower inference latency compared to control plane-based approaches, and improves classification accuracy by up to 21% over SOTA.

This journal version extends our previous conference paper [16] in three ways. First, it adds a formal analysis of the probabilistic token model and explains how rate-controlled feature export preserves fairness under heterogeneous traffic. Second, it gives substantially more implementation detail for the FPGA Model Engine, including streaming dataflow, CNN/RNN processing paths, HLS examples, INT8 quantization, and mixed-precision computation. Third, it expands the stress-test, latency methodology, and design-space positioning, especially the comparison with switch-only DNN systems and Taurus-style per-packet ML architectures.

II. Background and Motivation

This section reviews the evolution of DNN models for networking, examines the limitations of programmable switches, highlights the promise of FPGA-assisted networking, and outlines the integration challenges that motivate our hybrid system design.

DNN Learning Models for Networking. Classification tasks such as malicious traffic detection [1], [2], [3], [4] and application traffic classification [3], [5], [6] are fundamental for network intelligence. Prior work has demonstrated the feasibility of in-network AI inference by deploying lightweight models, such as decision trees [4], random forests [3], XGBoost [3], binarized MLPs [11], and simplified RNNs [6], on programmable switches and SmartNICs. While these models enable practical deployment, their simplified representations can limit accuracy on complex or dynamic traffic patterns.In contrast, more complex DNNs offer stronger nonlinear modeling capabilities and can adapt more effectively to evolving network conditions, leading to improved accuracy on diverse networking tasks. As a result, supporting DNN inference in network devices has the potential to further enhance the flexibility and precision of in-network intelligence, especially in scenarios where lightweight models face limitations.

Programmable Network Data Plane Limitations. Despite the promise of in-network DNN inference, the resource constraints of programmable data planes present significant challenges for deploying sophisticated models. PISA-based architectures are effective for basic machine learning tasks [3], [4], but encounter fundamental limitations with complex neural networks. The PISA instruction set supports only simple

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:09:32 UTC from IEEE Xplore. Restrictions apply.

operations (e.g., addition, subtraction, bit-shifting, and logical operations), and lacks native support for floating-point arithmetic, multiplication, division, and complex conditionals, operations essential for modern DNNs. Even when neural networks are quantized or translated into match-action-friendly forms, their scalability remains bounded by pipeline stages, table resources, and the limited arithmetic supported by switch ASICs.Hardware resources are also limited: for example, commercial switches such as Barefoot Tofino 1 offer only 12 pipeline stages and constrained memory resources (120 Mbit SRAM, 6.2 Mbit TCAM) [6], which are insufficient for parameter-rich neural networks. Furthermore, restrictions such as atomic register access prevent efficient implementation of iterative computations required by DNNs. Collectively, these constraints make it impractical to support DNN inference on programmable switches while maintaining line-rate performance, underscoring the need for alternative approaches to bring advanced neural network capabilities into the data plane.

FPGA-assisted Networking. FPGAs offer a promising solution to the computational limitations of programmable data planes while preserving high-performance networking capabilities. These reconfigurable platforms can interface directly with programmable switch ASICs, providing microsecond-level latency for DNN-based processing via customized hardware acceleration.[1] FPGAs strike a favorable balance among energy efficiency, computational flexibility, and performance, surpassing CPUs in energy efficiency and providing more versatile analytical capabilities than fixed-function switch ASICs [17]. Their support for partial dynamic reconfiguration enables network operators to update analytical functions without disrupting service [18], offering valuable operational flexibility. In hybrid architectures with programmable switch ASICs, these advantages are particularly valuable: conventional packet processing remains in the high-speed switch pipeline, while computation-intensive neural network inference is offloaded to specialized FPGA hardware. This clear division of labor enables the practical deployment of DNN models even in resource-constrained network environments.

Mismatch between FPGA and Switch. Despite their complementary capabilities, fundamental architectural differences between programmable switches and FPGAs present significant integration challenges. The two components operate in fundamentally different ways: switches process packets in deterministic pipelines with nanosecond precision, whereas FPGAs perform variable-latency neural computations requiring sophisticated synchronization. More critically, there is a substantial throughput gap between these chips: modern programmable switches operate at multi-terabit-per-second rates [14], while even the latest commercial FPGAs sustain only hundreds of Gbps [19]. This order-of-magnitude disparity makes it infeasible to offload all switch traffic directly to FPGAs for processing. As a result, a central challenge in FPGA-assisted networking is to export only the feature information that benefits from deep processing, while controlling the export rate so that FPGA inference does not stall switch forwarding.

[1]An FPGA can be directly connected to a switching ASIC as an external module, as implemented in our system; this approach also applies to switching ASICs other than Tofino.

Motivation. Current research in-network traffic analysis mainly follows two directions: adapting machine learning models to fit data plane constraints (e.g., transforming decision trees into match-action table representations) [3], [4], [6], [20], [21], or sampling data for offline analysis on servers [5], [22], [23]. However, both approaches inevitably compromise either model accuracy or introduce significant latency, and thus fall short of modern network requirements. The core challenge is to deploy powerful deep neural networks (DNNs) in high-speed network environments. While DNNs excel at traffic classification and anomaly detection, they face a critical bottleneck: programmable switches offer limited computational resources and restricted instruction sets, making complex DNN execution infeasible; meanwhile, hardware accelerators such as FPGAs, though efficient for neural network computation, provide processing bandwidth orders of magnitude lower than the Tbps-level throughput of switches. Therefore, a practical switch–FPGA design must decide which feature windows should reach the FPGA and must regulate this traffic according to the instantaneous load of both chips.

Goals. Our research aims to develop FENIX, an intelligent network data plane system that simultaneously achieves three core technical goals:

- **Low latency:** FENIX uses FPGA parallelism and direct interfacing with programmable switching ASICs to support neural network inference with microsecond-level latency, without involving software stacks or system buses. This hardware-based approach minimizes processing overhead and maximizes data throughput. As a result, FENIX ensures real-time responsiveness for latency-sensitive network applications.
- **High throughput:** FENIX keeps packet forwarding and feature extraction in the switch pipeline, while a rate-controlled feature export mechanism matches FPGA input traffic to accelerator capacity. This preserves line-rate forwarding and avoids overloading the channel between switch chips and FPGA.
- **High accuracy:** FENIX implements different DNNs (e.g. CNN, RNN) on FPGAs using fixed-point quantization, rather than simplified or binarized models. This approach maintains accuracy close to offline analysis and enables support for advanced pattern recognition and temporal analysis tasks.

III. FENIX Design Overview

Figure 2 shows the architecture of FENIX, which is designed to address two challenges in integrating deep learning inference with programmable switches. The first challenge is to support neural inference for fine-grained traffic control in the data plane. The second challenge is to bridge the throughput gap between programmable switching ASICs and FPGAs.

To address the throughput mismatch, FENIX uses a Data Engine on the switch ASIC to track flows, maintain feature history, and control feature export. The Flow Tracker records per-flow state and stores returned DNN labels. The Rate Limiter decides when a feature window should be exported to the FPGA, so not every packet is sent to the Model Engine.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:09:32 UTC from IEEE Xplore. Restrictions apply.

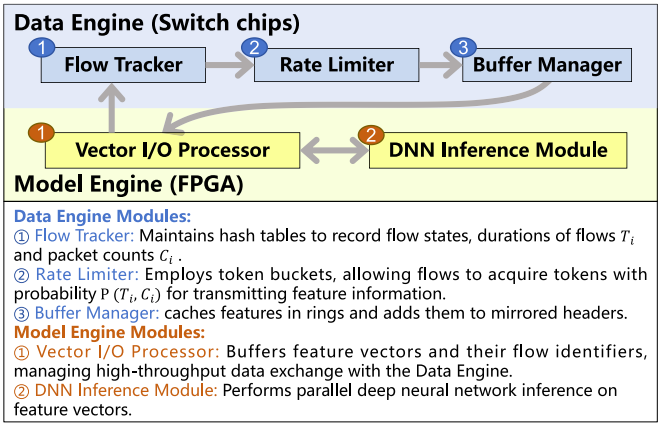


Fig. 2. The architecture of FENIX.

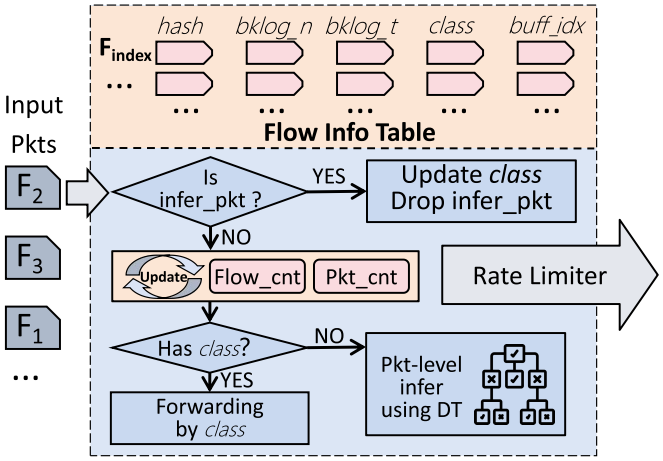


Fig. 3. Workflow of flow tracker.

The Buffer Manager maintains recent packet-level features and inserts them into mirrored packets for FPGA processing. FPGA inference is triggered only after rate-limiter admission; packets that are not exported continue through the switch pipeline using either a cached DNN label or the decision-tree fallback.

To support DNN inference on the data path, FENIX uses a Model Engine on the FPGA. The Vector I/O Processor separates the flow identifier and feature vector, preserves their correspondence, and returns inference results to the switch. The DNN Inference Module performs parallel neural-network inference on exported feature windows. Each returned result carries the corresponding flow identifier and refreshes the switch-side `class` field, allowing later packets from the same flow to reuse the latest DNN decision without waiting for another FPGA inference.

With this design, FENIX provides high-throughput, low-latency, and accurate DNN inference for in-network traffic analysis. The following sections detail each module design.

IV. Data Engine

As shown in Figure 2, the Data Engine of FENIX is implemented on the programmable switch ASIC and consists of three modules. The Flow Tracker records per-flow state, including flow duration T_i, packet count C_i, feature-buffer index, and cached classification result. The Rate Limiter uses a probabilistic token-bucket mechanism to control feature export according to $P(T_i, C_i)$, preventing the FPGA from being overwhelmed by excessive feature traffic. The Buffer Manager stores recent packet-level features in a per-flow ring buffer and exports them through mirrored packet headers. DNN execution therefore proceeds asynchronously: packets continue to be forwarded by the switch while selected feature windows are processed by the FPGA.

A. Flow Tracker

Overview. The Flow Tracker tracks network flows and makes per-flow forwarding decisions. As illustrated in Figure 3, it maintains a Flow Info Table in the switch's Static Random-Access Memory (SRAM). Each entry is indexed by a truncated hash of the packet five-tuple, including source IP, destination IP, source port, destination port, and protocol. For each flow, the table records the flow hash value (`hash`) for detecting new flows and hash collisions, backlog packet count (`bklog_n`), backlog timestamp (`bklog_t`), cached classification result (`class`), and buffer index (`buff_idx`). In the prototype, the switch-side flow table supports 65,535 usable entries and uses a 16-bit index derived from the low 16 bits of a 32-bit five-tuple hash.

Upon packet arrival, the Flow Tracker first checks whether the packet carries an inference result returned from the Model Engine. If so, it uses the flow identifier to locate the corresponding entry and updates the cached `class` field. For normal packets, the Flow Tracker checks whether a cached DNN result already exists. Packets with an existing result are forwarded according to the cached class. Packets without a returned DNN result use a lightweight decision tree on the switch ASIC as a provisional fallback.

Inference Trigger and Granularity. The decision tree is used only before a DNN result is available for a flow. For a new or unresolved flow, packets are first processed by the decision tree while the Rate Limiter decides whether to export a feature window to the FPGA. Once a DNN result returns, the Flow Tracker stores it in the `class` field and uses it for later packets of the same flow. If additional feature windows from the same flow are admitted later, the returned DNN result refreshes the cached class.

Online processing and evaluation granularity are distinct in FENIX. Both flow-level and packet/window-level DNN variants use exported packet-length and inter-packet-delay (IPD) feature windows as inputs. The online forwarding path does not require every packet to be exported to the FPGA. Instead, the switch exports only admitted feature windows, caches returned labels per flow, and forwards later packets using the latest cached decision. Flow-level evaluation reports one label per flow, whereas packet/window-level evaluation reports the prediction quality of each admitted feature window.

Flow Counting Mechanism. We define the start time of a flow as the instant when its first packet arrives at the switch, and count the number of flows that send packets within each

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:09:32 UTC from IEEE Xplore. Restrictions apply.

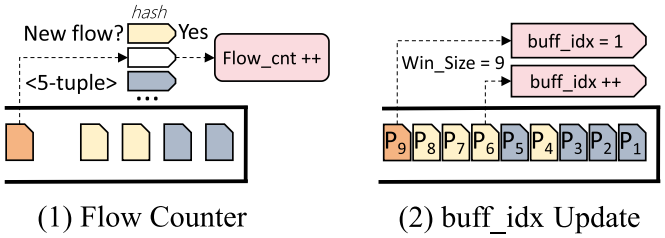


Fig. 4. Details in flow tracker.

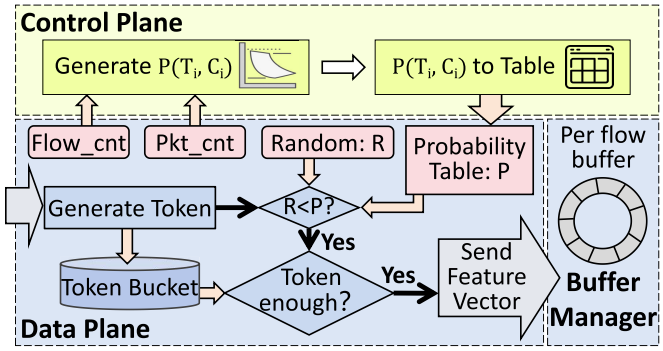


Fig. 5. Workflow of rate limiter.

timeout interval T_w. This statistical approach focuses on newly arrived flows within each interval and helps mitigate bias caused by missing exact flow start or end times [24]. As shown in Figure 4 (1), the flow counter detects new flows by checking hash registers associated with each flow. Upon detecting a new flow, the counter increments n by 1. At the end of each T_w period, the control plane resets both the hash registers and the flow count n to start a new counting cycle.

Buffer Index Update. As depicted in Figure 4 (2), the data plane cannot directly perform modulo operations for each packet. We therefore maintain a separate buffer index (`buff_idx`) for each flow. This index increments with each packet and resets to 1 when reaching the buffer size. The flow's packet counter (`Pkt_cnt`) independently records the total packet count. This design realizes a logical ring buffer in which new features overwrite old features in circular order.

B. Rate Limiter

Overview. Figure 5 illustrates the probabilistic rate limiter, which combines a dynamic probability model with a token-bucket mechanism. Its goal is to regulate feature export from the Data Engine to the Model Engine while preserving inference opportunities across flows.

The rate limiter maintains global traffic statistics within each timing window T_w, including `Flow_cnt` and `Pkt_cnt`. Standard work-conserving algorithms such as WFQ are mature mechanisms for inter-flow fairness, but they are not well suited to this setting. Allocating physical queues per flow is infeasible in the switch pipeline, and approximate queueing schemes cannot precisely match the bandwidth gap between the switch and FPGA. Our probabilistic token-bucket mechanism provides a hardware-friendly alternative. High-rate flows are more likely to fail the probability test, while lower-rate flows retain opportunities to export feature windows. The token bucket further bounds the total export rate to avoid overloading the FPGA.

TABLE I
VARIABLES OF TOKEN GENERATION PROBABILITY MODEL

Variable	Description
V	Token generation rate
Q	Global packet rate
N	Number of flows
T_i	Duration of backlogged packets
C_i	Number of backlogged packets
Q_i	Current flow packet rate
P_i	Probability of generating feature vector

Probability-Based Token Allocation Model. The rate limiter constructs a probability function $P(T_i, C_i)$ using real-time global traffic statistics, including the global flow count N and aggregate packet rate Q. Here, T_i denotes the elapsed time since flow i last transmitted its features, and C_i denotes the number of packets transmitted by flow i during this period. These variables approximate the current packet rate of flow i, $Q_i = C_i/T_i$. The token generation rate V is determined by the communication bandwidth B between engines, FPGA frequency F, and feature vector width W:

$$V = \min(F, B/W) \tag{1}$$

Token allocation is guided by two criteria.

Criterion 1: In the idealized scenario where all flows operate at equal rates, each flow should receive tokens at an average interval of $\frac{N}{V}$.

Criterion 2: In practical scenarios with heterogeneous rates, allocation should be proportional to flow speed. Thus, the expected token interval for flow i should approach $\frac{Q}{Q_i V}$, where $Q_i = \frac{C_i}{T_i}$.

Based on these criteria, we propose the following piecewise probability model:

$$P_i(T_i, C_i) = \begin{cases} \dfrac{C_i(VT_i - N)}{QT_i - NC_i}, & \text{if } T_i \in \left[\frac{N}{V}, \frac{Q}{Q_i V}\right], \left(\frac{N}{V} < \frac{Q}{Q_i V}\right) \\ \dfrac{T_i(VC_i - Q)}{NC_i - QT_i}, & \text{if } T_i \in \left[\frac{Q}{Q_i V}, \frac{N}{V}\right], \left(\frac{N}{V} > \frac{Q}{Q_i V}\right) \\ 1, & \text{if } QT_i = NC_i \text{ and } T_i \geq \frac{N}{V} \\ 0, & \text{if } QT_i = NC_i \text{ and } T_i < \frac{N}{V} \end{cases} \tag{2}$$

To validate the model, we analyze the expected token interval. For any flow i, the time-dependent sending probability over an interval $[t_s, t_e]$ can be represented as a cumulative distribution function:

$$P(t) = \frac{t - t_s}{t_e - t_s} \quad (t \in [t_s, t_e]) \tag{3}$$

The expected value of transmission time is:

$$E(t) = \int_{t_s}^{t_e} t \cdot f(t) dt \tag{4}$$

where $f(t)$ is the probability density function, $f(t) = P'(t) = \frac{1}{t_e - t_s}$. This gives us $E(t) = \frac{t_s + t_e}{2}$.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:09:32 UTC from IEEE Xplore. Restrictions apply.

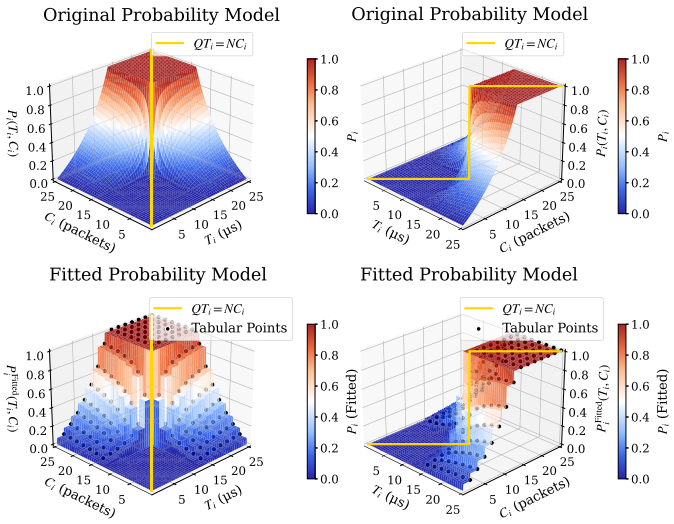


Fig. 6. Probability curves of token generation model.

For both cases ($\frac{N}{V} < \frac{Q}{Q_i V}$ and $\frac{N}{V} > \frac{Q}{Q_i V}$), the expected period for flow i is:

$$E_i = \frac{Q_i N + Q}{2 Q_i V} \tag{5}$$

Taking the average of expected periods across all flows weighted by their packet rates, we have:

$$E = \frac{\sum_{i=1}^{N} Q_i \cdot E_i}{Q} \tag{6a}$$

$$= \frac{\sum_{i=1}^{N} Q_i \cdot \frac{Q_i N + Q}{2 Q_i V}}{Q} \tag{6b}$$

$$= \frac{N \sum_{i=1}^{N} Q_i + Q \sum_{i=1}^{N} 1}{2 Q V} \tag{6c}$$

$$= \frac{NQ + QN}{2QV} \tag{6d}$$

$$= \frac{N}{V} \tag{6e}$$

In this derivation, Eq. (6a) computes the weighted average of individual expected periods E_i where weights are proportional to flow rates Q_i and $Q = \sum_{i=1}^{N} Q_i$ is the total packet rate. Eq. (6b) substitutes the expression for E_i from Eq. (5). Eq. (6c) expands the summation by distributing terms. Eq. (6d) applies the identities $\sum_{i=1}^{N} Q_i = Q$ and $\sum_{i=1}^{N} 1 = N$. Finally, Eq. (6e) cancels the common factor Q to obtain the overall expected interval $\frac{N}{V}$.

Therefore, the model achieves fair token allocation on average. High-rate flows obtain proportionally more opportunities, while the overall expected interval remains $\frac{N}{V}$. This keeps the aggregate export rate aligned with the Model Engine capacity.

Probability Model Deployment. To demonstrate the probability model, Figure 6 plots the function curves under representative network settings. The illustrated scenario contains 1000 concurrent flows, a model-engine processing rate of 75 Mpps, and an aggregate network throughput of 1000 Mpps. Since the data plane cannot compute the probability expression directly, the control plane precomputes probability ranges and installs them into the switch.

Algorithm 1 Token Bucket Algorithm for Rate Limiter

```
if T_last = 0 then
    T_last ← T_now, gap ← 0                ▷Initialize for first packet
else
    gap ← T_now − T_last, T_last ← T_now   ▷Calculate time gap
end if
rand ← Random(), prob ← LookupProbability()
bucket ← bucket + gap                      ▷Refill tokens
if rand < prob then                        ▷Selected for sampling
    if bucket ≥ cost then
        bucket ← bucket − cost             ▷Consume token
        SendFeatureVector()
    end if
end if
```

Since the switch pipeline cannot evaluate Eq. (2) directly, FENIX deploys the probability model as control-plane-generated range-match rules. At runtime, the data plane maps the current (T_i, C_i) state to a probability threshold, compares this threshold with a switch-generated random value, and exports a feature window only if the probability test and token check both succeed. The concrete table size, range granularity, threshold width, random-number source, and token cost used in our prototype are described in Section VI.

Token Bucket Algorithm Implementation. In our data plane design, the rate limiter tracks three state variables within a custom timing window T_w: the number of active flows, the total packet count, and the last packet arrival time T_{last}. Algorithm 1 summarizes the abstract token-bucket control logic. Upon packet arrival, the algorithm computes the timestamp gap between T_{now} and T_{last} to determine token replenishment. It then retrieves the probability value and compares it with a random value. If the probability test passes and the token bucket contains enough tokens, the bucket consumes the required cost and exports the feature vector. In the prototype, this abstract cost corresponds to `PER_TOKEN = 10` tokens.

Discussion. The Rate Limiter provides two safeguards. First, the probability model reduces excessive exports from high-rate flows while preserving inference opportunities for lower-rate flows. Second, the token bucket bounds the aggregate export rate so that the Model Engine is not overloaded. The probability model is computed in the control plane and deployed to the data plane as range-match rules, which keeps the runtime logic compatible with switch-pipeline constraints.

C. Buffer Manager

Ring Buffer Design. Inspired by BoS [6], the Buffer Manager assigns a dedicated ring buffer to each flow. The ring buffer stores recent packet-level features until the Rate Limiter admits a feature export. In the prototype, each flow maintains an eight-entry history for packet lengths and inter-packet delays, denoted as F_1–F_8. The current packet feature is stored in metadata and appended as the ninth feature position F_9 before export.This design preserves short-term temporal context within each flow while keeping the feature representation fixed-size and switch-friendly.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:09:32 UTC from IEEE Xplore. Restrictions apply.

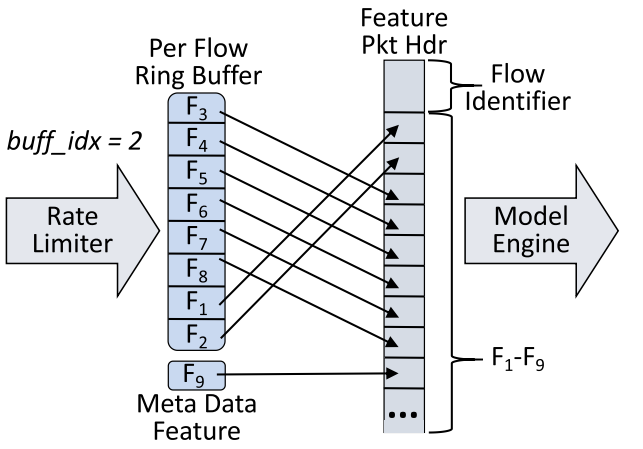


Fig. 7. Mechanism of buffer manager.

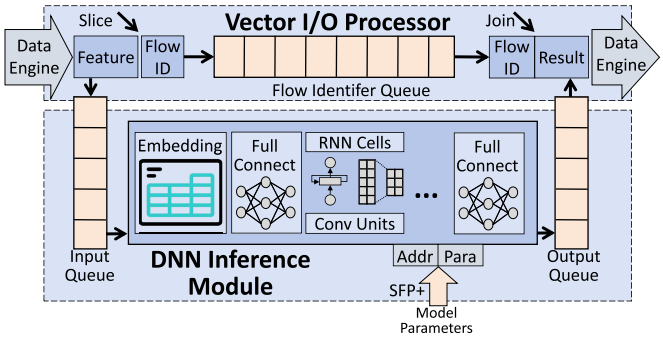


Fig. 8. Workflow of model engine.

When a buffer reaches capacity, it operates in circular order and overwrites the oldest feature with the newest one. This keeps the buffer aligned with recent traffic behavior and avoids unbounded per-flow storage.

Feature Transfer Process. The Buffer Manager works with the Rate Limiter to export selected feature windows. As shown in Figure 7, when the Rate Limiter admits an export, the Buffer Manager reads the `buff_idx` value from the Flow Tracker and extracts the corresponding feature sequence from the per-flow ring buffer. It then appends the current packet feature from metadata and assembles the resulting nine-position feature window into a mirrored packet header. The mirrored packet also carries the flow identifier, allowing the Model Engine to bind the returned DNN result to the correct flow entry.

V. Model Engine

The Model Engine performs low-latency inference on traffic features from the Data Engine and is implemented on an FPGA. As shown in Figure 8, it contains two components: a Vector I/O Processor and a DNN Inference Module. Upon receiving a mirrored packet, the Vector I/O Processor separates the flow identifier from the feature vector. The flow identifier is stored in a Flow Identifier Queue, while the feature vector is forwarded to the DNN Inference Module.

After inference completes, the result is placed in the Output Queue. The Vector I/O Processor pairs the inference result

```
1.  //HLS implementation of the Convolution Layer
2.  void Conv_Layer_Stream(
3.      hls::stream<input_t>& in_stream,
4.      hls::stream<output_t>& out_stream
5.  ) {
6.      input_t in_data = in_stream.read();
7.      output_t out_data;
8.
9.      // Parallel computation across output channels
10.     for (int c = 0; c < NUM_FILTERS; ++c) {
11.         #pragma HLS UNROLL
12.         dtype sum = conv_bias[c];
13.
14.         // Sliding window convolution
15.         for (int h = 0; h < KERNEL_H; ++h) {
16.             for (int w = 0; w < KERNEL_W; ++w) {
17.                 sum += in_data.val[h][w] * conv_weight[c][h][w]
18.             }
19.         }
20.
21.         // ReLU activation
22.         out_data.val[c] = (sum > 0) ? sum : 0;
23.     }
24.     out_stream.write(out_data);
25. }
```

Fig. 9. HLS implementation of the convolutional layer with parallel filter computation and ReLU activation.

with the corresponding flow identifier from the Flow Identifier Queue, constructs a return packet, and sends it back to the Data Engine. The switch then updates the `class` field of the matching flow entry. This design preserves the correspondence between exported feature windows and returned DNN labels throughout asynchronous FPGA processing.

A. Vector I/O Processor

The Vector I/O Processor parses incoming mirrored packets to extract the flow identifier and feature vector. It uses a FIFO queue to preserve the order of flow identifiers. The DNN Inference Module uses asynchronous FIFO queues for both input and output, allowing data to pass between modules that operate under different clock domains. This queue-based design decouples switch-FPGA communication from DNN execution and improves robustness under bursty feature exports.

During operation, the Vector I/O Processor monitors both the flow-identifier FIFO and the DNN output FIFO. When both queues contain valid entries, it dequeues the head elements, combines the flow identifier with the inference result, and sends the return packet to the Data Engine. For example, when a returned DNN result identifies a flow as malicious, the switch records this result in the flow's `class` field. Later packets from the same flow can then use the cached decision immediately.

B. DNN Inference Module

The DNN Inference Module performs neural-network inference on the FPGA, as illustrated in Figures 9 and 10. The core computation uses a systolic array optimized for INT8 operations. The implementation supports embedding

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:09:32 UTC from IEEE Xplore. Restrictions apply.

```
// HLS implementation of the Recurrent Layer
void RNN_Layer_Stream(
    hls::stream<vec_t>& x_in,
    hls::stream<vec_t>& h_out
) {
    vec_t h_state = 0; // Initialize hidden state

    for(unsigned int t = 0; t < SEQ_LEN; t++) {
        #pragma HLS PIPELINE II=1
        vec_t x = x_in.read();
        vec_t h_next;

        // Compute next hidden state
        for (int o = 0; o < HIDDEN_DIM; ++o) {
            #pragma HLS UNROLL
            dtype sum = rnn_bias[o];

            // Input-to-Hidden
            for (int i = 0; i < INPUT_DIM; ++i)
                sum += x.val[i] * w_ih[o][i];

            // Hidden-to-Hidden (Temporal dependency)
            for (int h = 0; h < HIDDEN_DIM; ++h)
                sum += h_state.val[h] * w_hh[o][h];

            h_next.val[o] = hls::tanh(sum);
        }

        h_state = h_next; // Update state register
        h_out.write(h_state);
    }
}
```

Fig. 10. HLS implementation of the recurrent layer with temporal state feedback for sequence modeling.

lookups mapped to LUTs, fully connected (FC) layers, convolutional layers, and recurrent layers. These layers are composed according to the configured model architecture, with layer parameters and feature dimensions fixed at synthesis time.

Figure 9 illustrates the convolutional layer implementation using stream-based dataflow. The design fully unrolls computation across output channels to exploit spatial parallelism, while sliding-window convolution is performed over the kernel dimensions. ReLU activation is applied inline after accumulation to reduce memory access. The actual system uses parallel filter banks with multiple kernel sizes to capture multi-scale spatial patterns.

Figure 10 shows the recurrent layer design with temporal state feedback. The implementation pipelines the temporal loop with an initiation interval of 1 cycle and fully unrolls hidden-dimension computation to increase throughput. The hidden state is maintained in registers and updated at each time step, enabling the model to capture sequential dependencies in traffic features.

To improve numerical robustness under INT8 quantization, we use mixed-precision compute and accumulation in the systolic array. Multiplications use 8-bit operands, while partial sums and multiply-accumulate results are accumulated in 32-bit width to avoid overflow. After accumulation, results are re-quantized to 8 bits for later layers using per-layer scale-and-shift operations with rounding and saturation. This scheme balances accuracy and hardware efficiency.

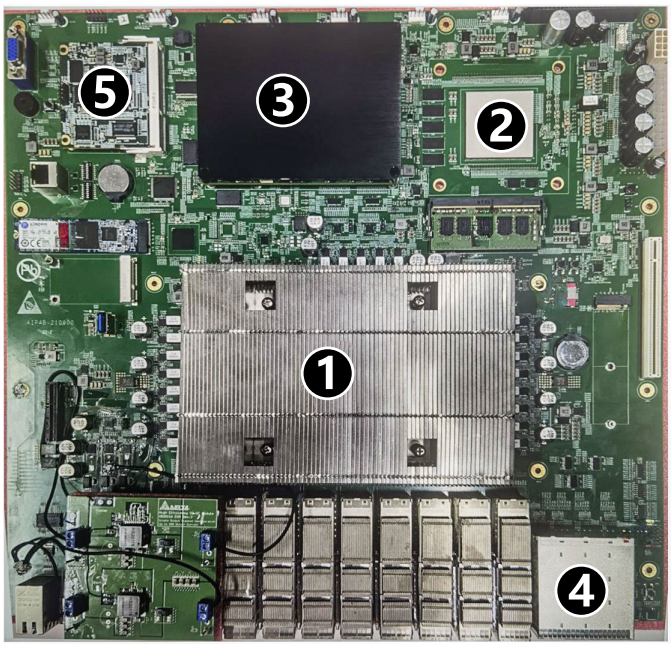


Fig. 11. Hardware platform of FENIX: The FPGA-Enhanced programmable switch implementation.

Model parameters, including weights and biases for all supported layers, are loaded from the host into on-chip memory through the network interface. During inference, feature vectors are processed in batches. Embedding lookups are performed first, followed by the configured FC, convolutional, and recurrent layers. Asynchronous FIFO queues decouple dataflow between modules and enable efficient pipelining.

VI. Implementation

Experimental Hardware. As shown in Figure 11, our experiments are conducted on a high-performance programmable switch platform that integrates both FPGA and Tofino chips on a single board. The hardware is implemented using a 22-layer high-performance printed circuit board (PCB), utilizing back-drilling technology to balance 100 Gbps high-speed interconnection performance and manufacturing costs. The switching system is equipped with a dedicated power control network, supporting a core operating current of up to 100 A and peak control currents up to 300 A, with precise impedance control to ensure reliable power delivery. The overall design process included approximately half a month for requirement analysis, one month for schematic design, two months for PCB design, and one and a half months for PCB manufacturing, soldering, and prototype hardware debugging. The interface design and testing between FPGA and Tofino required an additional month, with software driver development and system-level integration carried out in parallel.

Figure 11 presents a photograph of the hardware prototype, with the primary components highlighted as follows:

❶ **Tofino 2 switch chip** [13]: Featuring 20 MAU stages, 200 Mbits of SRAM, and 10.3 Mbits of TCAM per pipeline. We configured this chip to efficiently implement our Data Engine modules while balancing resource utilization across

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:09:32 UTC from IEEE Xplore. Restrictions apply.

pipeline stages. Multiple 100 Gbps port channels were designed to connect the Tofino chip to the FPGA, enabling high-bandwidth data exchange.

❷ **Xilinx ZU19EG FPGA chip** [19]: Integrates approximately 80 Mbits of on-chip memory and 1,143,450 logic elements. The FPGA resources were partitioned to accommodate the Model Engine's Vector I/O Processor and Neural Computing Array, while maintaining timing closure at high clock frequencies. The FPGA is directly connected to the Tofino chip through high-speed port channels, enabling low-latency communication.

❸ **Control plane CPU**: Programmed to manage the Tofino chip, responsible for configuration, monitoring, and probability model adjustments. It also reads runtime packet and flow counters, recomputes the rate-limiter probability table, and installs updated entries through BF Runtime.

❹ **Four front-panel FPGA ports**: Designed with configurable transceivers to operate in either 10G or 25G modes for external communication. The design process required careful signal integrity analysis and impedance matching.

❺ **BMC module**: Customized for hardware status monitoring and remote management of the platform, including thermal management for high-performance components.

Experimental Software. Our development process spans multiple hardware and software platforms and integrates several toolchains. For the Tofino chip, we used SDE version 9.8.0 for software development and debugging. For FPGA programming, we used Vivado and Vitis HLS 2024.1 [25]. The prototype includes approximately 1,500 lines of P4 code [7] for the Data Engine, covering the Flow Tracker, Rate Limiter, and Buffer Manager on the programmable switch ASIC. This code implements hash table maintenance, token bucket control, and feature cache rings. The FPGA-side Model Engine contains 1,000 lines of HLS code for the Vector I/O Processor and DNN Inference Module, supporting parallel operations and efficient matrix processing. The software toolchain includes 1,500 lines of Python code and 1,000 lines of C++ code for model training, testing, and quantization under hardware resource constraints. It also includes 300 lines of Python code for calculating control-plane probability models, which guide feature transmission and adaptive flow management. In addition, we developed an 800-line Python simulator to evaluate and validate FENIX's performance and scalability under high-throughput and large-scale network scenarios.

In the prototype, the switch-side flow table supports 65,535 usable entries and uses a 16-bit index derived from the low 16 bits of a 32-bit five-tuple hash. For each flow, the Buffer Manager keeps the previous eight packet-length and inter-packet-delay (IPD) features, and appends the current packet as the ninth feature position before export. The Rate Limiter implements the probability model with a range-match table that provides 1,024 available entries. The control plane installs 20×20 active probability ranges over $T_i \in [1, 65535]$ timestamp units and $C_i \in [1, 320]$ packets, and refreshes these ranges once per second. Each active range stores a 16-bit fixed-point threshold τ. After clipping $P(T_i, C_i)$ to $[0, 1]$, the control plane shifts the probability into the 16-bit random-value domain and stores $\tau = \lfloor P(T_i, C_i)(2^{16} - 1) \rfloor$. At runtime, the Tofino `Random<bit<16>>()` extern generates a 16-bit value r, and the P4 action uses this threshold to realize the admission test $r \leq \tau$. A feature window is exported only when this probability test succeeds and the token register can consume `PER_TOKEN = 10` tokens; otherwise, the data plane only refreshes the token state using the elapsed timestamp gap.

Model Training and Quantization. We trained several neural network architectures, including Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), for intelligent traffic analysis and classification tasks. Our approach leverages universal and protocol-agnostic traffic features: sequences of raw packet lengths and inter-packet arrival times, which capture the temporal patterns of network flows without requiring any protocol-specific information. For feature extraction, we implemented a software-based sliding window mechanism to process raw PCAP files and generate feature vectors, each consisting of a sequence of packet length and timing data from a flow, paired with the corresponding class label for supervised learning.

Our model architectures generally include normalization layers to standardize input features, followed by either convolutional layers (for CNNs) or recurrent units (for RNNs) to capture temporal dependencies. These are further connected to fully connected layers that map extracted features to output classes, with ReLU activation functions introducing non-linearity. To address class imbalance in-network traffic datasets, we applied both oversampling and undersampling techniques during preprocessing, thereby preventing overfitting to dominant classes and improving the representation of minority classes. This comprehensive approach results in more robust models with higher accuracy and better generalization for identifying diverse traffic types.

After training, we performed quantization using Vitis-AI [26] tools to convert floating-point model parameters to 8-bit integer (INT-8) format for efficient hardware deployment. The quantization process assigns different decimal point positions to different layers, based on their activation distributions, to preserve model accuracy. Our experimental evaluation shows that quantization substantially reduces computation and storage requirements while maintaining classification accuracy, with only negligible performance degradation. The quantized models are thus well-suited for deployment on resource-constrained FPGA platforms, enabling high-throughput, low-latency inference without sacrificing classification quality or robustness.

Model Engine FPGA Implementation. The model engine adopts a streaming dataflow architecture where computational stages are interconnected through FIFO channels. Only feature-carrying mirrored packets admitted by the rate limiter enter the FPGA inference pipeline; ordinary packets continue through the switch pipeline using either cached DNN results or the switch-side fallback.The streaming pipeline achieves a throughput of one packet every three cycles at a 300 MHz operating frequency.

For different models, the front-end feature processing of the model inference module remains identical. Feature separation splits the 9×2 input matrix into length and IPD streams.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:09:32 UTC from IEEE Xplore. Restrictions apply.

The embedding layer converts these indices into dense vectors through lookup tables, and feature concatenation merges them into a unified representation for subsequent layers. Thus, each FPGA inference operates on one admitted nine-packet feature window rather than on every packet in the online traffic stream.

The model-specific processing paths are as follows: The CNN model treats the representation as a 2D feature map and applies parallel convolutional filters followed by global average pooling and a fully connected classifier. The RNN model spatially unrolls recurrent computation across the nine time steps, propagates hidden states through FIFO channels, and feeds the final hidden state into the classifier.

The model output is fed into a fully connected classification layer that maps the multi-dimensional representation to per-class logit values. A log-softmax module then computes normalized log-probabilities for each admitted feature window. The Vector I/O Processor pairs each output with the corresponding flow identifier and returns the result to the switch, where it refreshes the cached `class` field for subsequent packets of the same flow.The system design includes a parameter update interface that streams new weights into the FPGA during reconfiguration to update embedding tables, transformation matrices, and bias vectors. This datapath reduces parameter loading overhead and shortens reconfiguration and update latency, enabling efficient parameter refresh.

VII. Evaluation

Our evaluation addresses several key questions: (i) What classification accuracy can FENIX achieve on real traffic datasets, and how does it compare to other methods? (ii) What computational resources are required by FENIX, including both programmable switching ASICs during feature extraction and FPGA resources? (iii) How is FENIX performance affected under different scales of concurrent flows and throughput? (iv) As an asynchronous hybrid system, does FENIX introduce noticeable latency overhead when using FPGA for inference?

To answer these questions, we implemented FENIX and other baselines on an FPGA-enhanced programmable switch using P4 [7] and HLS [25] (Details in Section § VI). We conducted comprehensive comparative evaluations across multiple dimensions, including accuracy, resource utilization, scalability, and latency. For testing purposes, we used publicly available datasets for VPN encrypted traffic classification and malware detection. These datasets provide diverse traffic patterns and represent real-world network scenarios, enabling us to thoroughly assess FENIX's capabilities in practical deployment environments. The evaluation setup allows us to systematically measure the trade-offs between classification performance and hardware efficiency.

A. Methodology

Testbed Setup. We implemented FENIX on an FPGA-enhanced programmable switch (Details in Section § VI) using P4 and HLS, connected to a high-performance server equipped with dual Intel Xeon 5418Y processors and 512GB of memory. The server is equipped with two NVIDIA MCX75310AAS-NEAT 400G network interface cards, with one dedicated to traffic generation and the other to packet reception. The sending NIC generates synthetic traffic and replays pcap files using DPDK (v23.11.3) pktgen [29], while the receiving NIC collects and analyzes the processed packets using the DPDK environment.

Schemes compared. We compared FENIX with the following 9 schemes, including both flow-level and packet-level comparisons. FlowLens is compared at the flow level because it performs control-plane inference and outputs flow decisions. For FENIX-CNN-FLOW and FENIX-RNN-FLOW, we obtain flow labels by majority voting over packet predictions. Packet-level schemes are evaluated by counting the prediction assigned to each packet.

(a) FENIX-CNN-FLOW: We implemented a CNN classifier with 3 convolutional layers (64, 128, 256 filters) and 2 fully connected layers (512, 256 neurons). For flow-level evaluation, each packet is classified by the CNN, and the final flow label is obtained by majority voting over all packet predictions in the flow.

(b) FENIX-RNN-FLOW: We implemented an RNN classifier with a single custom RNN cell (128 units) and a dense output layer. For flow-level evaluation, the model classifies packets using packet length and IPD features, and the final flow label is obtained by majority voting over all packet predictions in the flow.

(c) FlowLens: [5] We implemented packet statistics in Tofino using FlowLens' FMA code with control plane inference. Our XGBoost model used default parameters [30], with collection windows adjusted based on throughput.

(d) FENIX-CNN-PKT: We implemented a packet-level CNN with the same architecture as FENIX-CNN-FLOW. For packet-level evaluation, each packet prediction is counted independently, and accuracy is computed from the per-packet classification results.

(e) FENIX-RNN-PKT: We implemented a packet-level RNN with the same architecture as FENIX-RNN-FLOW. For packet-level evaluation, each packet is classified independently using packet length and IPD features, and accuracy is computed from the per-packet classification results.

(f) Netbeacon: [3] We implemented multi-phase tree models. Each phase uses a Random Forest (3 trees, depth 7) matching their configuration.

(g) Leo: [4] We implemented a decision tree (max depth 22, up to 1024 leaf nodes) on switches using packet length extremes and cumulative flow length.

(h) BoS: [6] We implemented the largest variant with a binarized GRU network (9-bit hidden states), 8 GRU units, 6-bit embeddings, and complete feature embedding-GRU-output architecture.

(i) N3IC: [11] We implemented a binary MLP on SmartNIC using flow-level and packet-level features with hidden layers [128, 64, 10]. Due to hardware constraints, we simulated switch-side logic and inference in software.

Tasks. We use the following tasks to evaluate FENIX, as summarized in Table II. (i) Encrypted Traffic Classification on VPN: This task aims to classify network traffic that has been encrypted by VPNs. We use the ISCXVPN2016 dataset [27]. (ii) Malware Identification: This task distinguishes between

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:09:32 UTC from IEEE Xplore. Restrictions apply.

TABLE II
EXPERIMENTAL SETTINGS

Dataset (Task)	ISCXVPN2016 [27]	USTC-TFC [28]
Training Flows	29,295	101,789
Test Flows	7,328	25,455
Number of Classes	7	12
Class Ratio	11:4:13:10: 18:128:1	92:10:4:14:17:23: 105:1:16:132:27:1
Optimizer	AdamW	AdamW
Learning Rate	0.01	0.005

traffic generated by benign applications and various types of malware. We use the USTC-TFC2016 dataset [28]. Note that model training is performed offline, and the FPGA is used exclusively for inference tasks.

Metrics. We report macro-F1 at the evaluation granularity of each method. FlowLens is evaluated with flow-level macro-F1 because it performs control-plane inference and outputs flow decisions. For FENIX-CNN-FLOW and FENIX-RNN-FLOW, flow labels are obtained by majority voting over packet predictions. For packet-level schemes, macro-F1 is computed over individual packet predictions. We also report class-wise Precision/Recall.

B. Classification Accuracy on Real Datasets

Table III presents a comprehensive comparison of accuracy and recall performance on encrypted traffic classification and malware detection. The flow-level columns compare FlowLens with FENIX-CNN-FLOW and FENIX-RNN-FLOW under flow-level labels, while the packet-level columns report per-packet prediction quality.Across all categories in both tasks, FENIX consistently demonstrates superior accuracy compared to other baselines. At the flow level, the macro-F1 score of FENIX reaches 0.890, which is very close to that of FlowLens (0.870), indicating that FENIX can achieve state-of-the-art detection performance while retaining the advantages of a neural network-based approach.

When evaluated at the packet level, FENIX outperforms all other compared methods across both tasks. On the malware detection benchmark, FENIX achieves a macro-F1 score of 0.907, greatly surpassing tree-based approaches such as NetBeacon (0.670) and showing a clear margin over advanced neural baselines like N3IC (0.858). Compared to other neural models, including Leo (0.741) and BoS (0.814), FENIX also demonstrates consistently higher accuracy.

This performance gap becomes especially pronounced in challenging multiclass scenarios, where traditional methods often suffer from error propagation or limited packet-level representational capacity. While BoS and Leo achieve moderate results, they still lag behind FENIX, highlighting the benefits of high-precision inference enabled by the FENIX architecture. Notably, BoS is constrained by model binarization and the limited size of each component that switching ASICs can accommodate, which reduces its accuracy. Tree-based methods like NetBeacon can only update predictions at discrete points, further limiting performance on fine-grained, packet-level tasks. Although FENIX does not perform continuous inference for every packet, its higher per-inference accuracy ensures better overall packet-level accuracy. This advantage is particularly evident in complex multiclass settings, where the CNN-based FENIX model achieves the highest classification performance among all tested methods.

C. Hardware Resource Utilization

Table IV compares the hardware resource overhead of several representative P4 systems on programmable switches. The results show that FENIX achieves relatively low resource usage across multiple dimensions. For instance, its SRAM consumption (12.9%) is much lower than FlowLens (34.2%), indicating that the introduction of FPGA effectively reduces the resource overhead of P4 systems. In terms of TCAM usage, FENIX also stands out, with only 4.4% overhead compared to NetBeacon's 18.8%, highlighting that FENIX avoids the trade-off of reducing one resource at the cost of sharply increasing another. Beyond memory, FENIX maintains efficient use of bus bandwidth and pipeline stages, remaining below or comparable to most other systems. This balanced and predictable resource profile is enabled by the decoupled DataEngine design, which keeps FENIX's overhead stable across different tasks and workloads.

Table V summarizes the FPGA resource utilization, where percentages represent the proportion of total available resources consumed by each module. The results show that LUTs and FFs are the primary resources consumed, particularly in the core computational components of the CNN and RNN modules. For example, the overall CNN module uses 38.4% of LUTs and 33.8% of FFs. While BRAM and DSP usage is higher in the overall modules (up to 7.1% and 8.1% for CNN, and 6.3% and 4.6% for RNN, respectively), it remains minimal within individual submodules, where most values are below 4%. This indicates that our design primarily leverages combinational and sequential logic to efficiently implement neural network inference on the FPGA, while memory and DSP consumption remains moderate. These FPGA-side results complement Taurus's ASIC-level resource results for data-plane ML [10]: Taurus reports that a full 12×10 MapReduce grid adds 3.8% chip-area overhead and 2.8% power overhead while sustaining 1 GPkt/s line rate, with representative KMeans, SVM, and DNN models adding 61 ns, 83 ns, and 221 ns latency, respectively. Taurus therefore quantifies the efficiency of placing ML operators directly into dedicated switch silicon, whereas FENIX targets a more deployable switch-FPGA split, where the main challenge is bridging line-rate packet processing and FPGA inference throughput without sharply increasing switch-side SRAM, TCAM, bus, or stage usage.Overall, the resource utilization is low, leaving ample headroom for further optimization and larger deployments.

D. Flow Count and Throughput Scalability

We conduct stress tests on FENIX under high-concurrency and high-throughput scenarios. Since all the original network

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:09:32 UTC from IEEE Xplore. Restrictions apply.

TABLE III
PERFORMANCE COMPARISON OF DIFFERENT METHODS ON ENCRYPTED TRAFFIC CLASSIFICATION AND MALWARE DETECTION

Class	FENIX $_{\text{F-CNN}}$	FENIX $_{\text{F-RNN}}$	FlowLens [5]	FENIX $_{\text{P-CNN}}$	FENIX $_{\text{P-RNN}}$	NetBeacon [3]	Leo [4]	BoS [6]	N3IC [11]
Encrypted Traffic Classification (ISCXVPN2016 [27])									
Chat	0.883/0.852	0.939/0.882	0.862/0.922	0.917/0.836	0.804/0.653	0.627/0.169	0.489/0.353	0.922/0.913	0.533/0.527
Email	0.924/0.834	0.944/0.924	0.888/0.821	0.862/0.882	0.893/0.746	0.230/0.321	0.333/0.019	0.932/0.925	0.349/0.353
File	0.879/0.849	0.923/0.912	0.860/0.889	0.886/0.797	0.889/0.844	0.861/0.701	0.815/0.720	0.915/0.922	0.820/0.848
P2P	0.977/0.963	0.976/0.988	0.923/0.913	0.911/0.914	0.932/0.947	0.861/0.908	0.853/0.867	0.925/0.917	0.905/0.892
Stream	0.902/0.968	0.919/0.973	0.966/0.959	0.877/0.965	0.894/0.969	0.890/0.976	0.850/0.944	0.916/0.908	0.886/0.938
Voip	0.989/0.995	0.992/0.996	0.998/0.995	0.999/0.998	0.999/0.999	0.986/0.993	0.994/0.997	0.829/0.723	0.994/0.993
Web	0.803/0.662	0.793/0.625	0.700/0.525	0.800/0.861	0.869/0.821	0.860/0.405	0.784/0.046	0.729/0.623	0.856/0.524
Macro-F1	0.890	0.912	0.870	0.892	0.873	0.658	0.578	0.863	0.738
Malware Detection (USTC-TFC [28])									
Cridex	0.999/1.000	0.999/1.000	0.999/0.998	0.999/1.000	0.996/1.000	0.933/0.997	0.984/0.995	0.983/0.996	0.866/0.861
FTP	0.999/0.998	0.999/0.998	0.999/0.998	0.997/1.000	0.993/1.000	0.993/0.485	0.928/0.986	0.986/0.791	0.848/0.853
Geodo	0.979/0.881	0.984/0.891	0.945/0.905	0.932/0.343	0.786/0.424	0.899/0.524	0.369/0.188	0.934/0.615	0.857/0.851
Htbot	0.962/0.987	0.959/0.989	0.964/0.984	0.932/0.986	0.938/0.982	0.830/0.782	0.897/0.935	0.919/0.937	0.871/0.867
Neris	0.921/0.594	0.888/0.732	0.902/0.817	0.766/0.473	0.736/0.575	0.665/0.469	0.584/0.453	0.761/0.610	0.852/0.847
Nsis-ay	0.985/0.970	0.971/0.982	0.985/0.986	0.959/0.968	0.962/0.972	0.984/0.912	0.918/0.960	0.963/0.968	0.860/0.855
Warcraft	0.998/0.999	0.996/0.994	0.995/0.993	1.000/1.000	0.999/1.000	0.890/1.000	0.995/0.997	0.956/0.998	0.855/0.859
Zeus	0.932/0.945	0.955/0.863	0.971/0.932	0.962/0.978	0.978/0.958	0.823/0.260	0.895/0.791	0.976/0.951	0.865/0.860
Virut	0.671/0.941	0.760/0.893	0.827/0.908	0.723/0.863	0.763/0.836	0.571/0.815	0.691/0.692	0.744/0.844	0.859/0.855
Weibo	0.684/0.822	0.702/0.927	0.745/0.804	0.655/0.821	0.685/0.817	0.649/0.789	0.653/0.708	0.652/0.837	0.862/0.859
Shifu	0.999/0.966	0.997/0.995	0.982/0.914	0.947/0.829	0.965/0.767	0.194/0.302	0.610/0.593	0.835/0.501	0.862/0.859
SMB	0.700/0.522	0.845/0.504	0.726/0.654	0.632/0.416	0.665/0.492	0.607/0.434	0.555/0.491	0.651/0.406	0.862/0.859
Macro-F1	0.887	0.901	0.914	0.907	0.838	0.670	0.741	0.814	0.858

Note: Class entries are Precision/Recall. Macro-F1 is reported at the corresponding evaluation granularity: FENIX $_{\text{F-CNN}}$, FENIX $_{\text{F-RNN}}$, and FlowLens use flow-level labels, while the remaining methods use packet-level predictions. For FENIX $_{\text{F-CNN}}$ and FENIX $_{\text{F-RNN}}$, flow labels are obtained by majority voting over packet predictions.

TABLE IV
P4 SYSTEMS RESOURCE OVERHEAD COMPARISON

System	SRAM	TCAM	Bus	Stage
FENIX	12.9%	4.4%	3.5%	9
FlowLens [5]	34.2%	0.0%	2.4%	9
BoS [6]	26.3%	6.3%	8.6%	12
Leo [4]	26.9%	9.0%	5.2%	12
NetBeacon [3]	11.6%	18.8%	6.4%	12

TABLE V
NEURAL NETWORK RESOURCE UTILIZATION (%)

Module	LUT	FF	BRAM	DSP
CNN (overall)	38.4%	33.8%	7.1%	8.1%
Embedding	4.2%	5.1%	0.5%	0.0%
Convolutional	25.6%	19.7%	4.0%	5.7%
FC	8.6%	9.0%	2.6%	2.4%
RNN (overall)	25.6%	31.2%	6.3%	4.6%
Embedding	4.2%	5.1%	0.5%	0.0%
Recurrent	15.8%	18.7%	3.6%	2.4%
FC	8.6%	9.2%	2.2%	2.2%
Vector I/O	6.0%	4.8%	0.3%	0.0%

trace files were collected in low-bandwidth networks (e.g., tens of Mbps), we generate high-throughput trace files by concurrently packaging a large number of flows (ensuring each flow has a unique identifier) and reassigning new timestamps to these packets.[2] We then replay these pcap files on our testbed to generate actual network traffic.

Figure 12 presents the results of the scalability test, where we progressively increase the flow concurrency until the physical bandwidth of the traffic generator's NIC is fully saturated. The results demonstrate that FENIX can easily handle this level of scale, as its macro-F1 scores remain nearly identical to those in Table III.

To evaluate larger scales, we build a simulator emulating FENIX's workflow. We configure maximum concurrent flow count based on switch chip registers and set inference latency according to measured results. By replicating the experimental settings of Table III and Figure 12, we validate the accuracy of the simulator, finding that the simulation results are almost the same as those from our testbed. Subsequently, we use the simulator to explore significantly larger scales, progressively increasing the flow concurrency to tens of thousands of flows per second and the aggregate throughput to the Tbps level. As shown in Figure 12, the macro-F1 score of FENIX experiences only a minor decrease, with a reduction of about 13.2% at the largest scale.

[2]We write the original timestamp into the packet header to handle changes caused by flow acceleration.

E. Latency Microbenchmark

We conduct a comprehensive latency breakdown comparing the control-plane inference approach (FlowLens [5]) with our

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:09:32 UTC from IEEE Xplore. Restrictions apply.

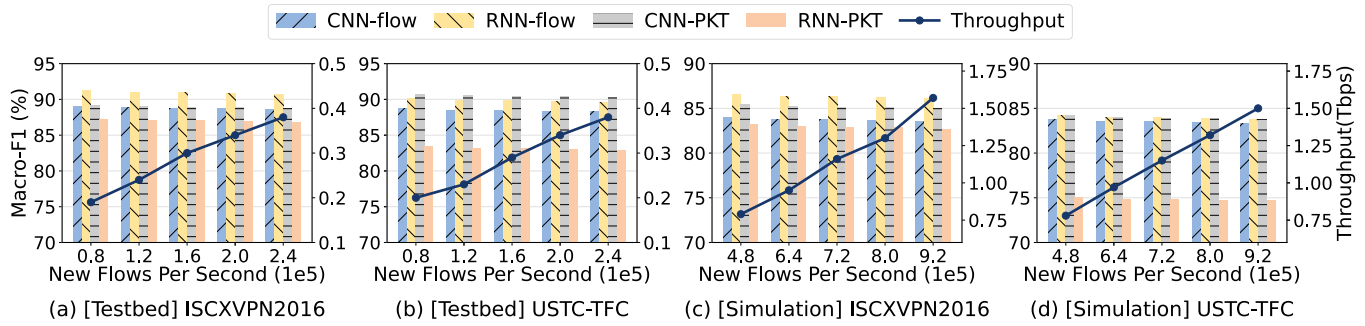


Fig. 12. Scaling test of FENIX.

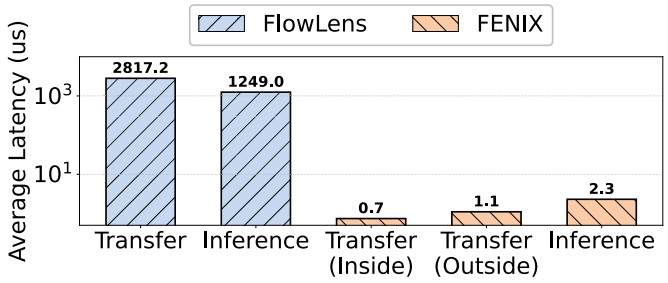


Fig. 13. Average latency compare of FENIX and FlowLens.

proposed FENIX, as summarized in Figure 13. To capture microsecond-scale delays, we use an RTT-based method: internal transmission is measured via PCB interconnects, and external transmission via optical modules. Inference latency reflects the total model execution time for each system.

The results show that FENIX significantly outperforms FlowLens across all latency components. FlowLens, which relies on general-purpose CPUs and control-plane communication, experiences transmission and inference delays in the millisecond range (e.g., 2.1 ms for transmission and 1.5 ms for inference). In contrast, FENIX benefits from a tightly integrated FPGA architecture, where the model and data engines communicate directly within the FPGA fabric. This design eliminates PCIe and memory bus overhead, resulting in sub-microsecond internal transmission and $1--3\mu$s external transmission.

Most importantly, FENIX reduces inference latency by nearly three orders of magnitude compared to FlowLens. While FlowLens requires over $1,000\mu$s for each inference, FENIX completes inference in just 1.2μs on average. This improvement comes from in-network acceleration on FPGA, which offloads model execution from the host CPU and enables efficient, microsecond-scale inference.

VIII. Discussion

Hardware Adaptability Beyond Tofino. FENIX is prototyped using a Tofino 2 switch and Xilinx ZU19EG FPGA, but its architecture is fundamentally portable. The discontinuation of Intel's Tofino line [31] does not threaten the broader applicability of FENIX, as the programmable switch ecosystem remains rich, including Juniper Trio [32], Broadcom Trident [33], runtime programmable switch designs [34], [35], [36], and RMT-based research designs such as dRMT [37] and Menshen [38]. Industry trends increasingly embrace intelligent data planes, as exemplified by Broadcom Trident 5's NetGNT, which integrates neural traffic analysis directly into switching silicon [33]. However, public information on such commercial products still lacks detailed specifications of AI accelerator capabilities and programming interfaces. FENIX therefore provides an open switch–FPGA reference architecture that can be adapted to future programmable switching platforms.

Prototyping for ASIC Evolution. Taurus [10] has set an important precedent for per-packet ML in the data plane by combining architectural extensions with switch/FPGA prototyping and ASIC synthesis.However, barriers to ASIC adoption remain high due to inflexible design cycles and prohibitive development costs. FENIX addresses this gap by providing a commercial, reprogrammable prototype that achieves high throughput and low latency at switch nodes, serving as both a practical solution for immediate deployment and a reference architecture for future ASIC designers. Different from Taurus, FENIX focuses on the system boundary between an existing programmable switch ASIC and an attached FPGA co-processor, including feature export, nonblocking switch processing, and result return to the data plane.The design principles, resource allocation strategies, and performance trade-offs explored in FENIX can inform the development of next-generation switching ASICs with integrated neural acceleration capabilities.

Automation and Future Directions. Recent frameworks such as Homunculus [39] highlight the growing need for automated ML pipelines and model deployment in data plane environments [40]. While FENIX currently requires expert-driven integration when migrating to new tasks or hardware platforms, its modular Model Engine design is well-suited for future automation. However, the current FPGA implementation has inherent limitations regarding flexibility. Layer parameters and feature dimensions are fixed during synthesis, meaning that DNN architecture updates typically require hardware re-synthesis. Extending FENIX with task-agnostic abstractions, automated switch/FPGA toolchains, and partial-reconfiguration-based model switching would lower deployment barriers and improve adaptability.

IX. Related Work

ML for Network Traffic Analysis. Machine learning has become a core tool for network traffic analysis, powering solutions for malicious traffic detection [1], [2], [41], [42],

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:09:32 UTC from IEEE Xplore. Restrictions apply.

website fingerprinting [43], [44], [45], and fine-grained traffic classification [46], [47]. The field is rapidly advancing toward sophisticated models and tasks, including traffic prediction and encrypted flow analysis [48], [49], [50]. These applications increasingly demand real-time inference capabilities to support timely security responses and dynamic traffic management. However, deploying such models at line rate in the data plane remains challenging due to hardware constraints, limited computational resources in network devices, and the tension between model complexity and inference latency.

Data Plane Device Roles and Limitations. Modern network infrastructures rely on programmable switches, SmartNICs, and FPGAs, each with distinct strengths and deployment scenarios. SmartNICs efficiently offload compute from servers for functions such as rate limiting, traffic analysis, and protocol acceleration [11], [51], [52], [53], [54], providing flexible processing at network endpoints but with limited visibility into transit traffic. Programmable switches provide unparalleled throughput and enable in-network tasks such as DDoS defense and real-time classification [3], [4], [6], [20], [21], [55], [56], [57], [58], [59], [60], [61], [62], though their rigid resource budgets and limited computational primitives constrain model complexity. FPGAs serve as both prototyping platforms and flexible data plane accelerators [10], [17], [18], [63], [64], [65], [66], [67], [68], [69], offering reconfigurability and computational capacity that complement ASIC-based switches.

Comparison and Positioning. Several commercial products integrate hardware accelerators with programmable switch ASICs, such as APS Networks' APS6120Q [70] and Cisco's smart switch series [71]. However, these systems typically encapsulate functionality behind proprietary network operating systems and do not publicly disclose accelerator-switch integration mechanisms, deployable model scales, resource allocation strategies, or open neural-inference evaluations. This opacity makes them difficult to use as general-purpose research platforms or rigorous baselines for in-network DNN inference.

Prior academic work has primarily taken two approaches to deploying ML in network data planes. One focuses on deploying quantized or highly compressed ML models directly in programmable switches [4], [20], [21], [60], [61], [62]. Recent systems such as Quark [72] and Pegasus [59] further develop this switch-only direction by mapping quantized neural inference into programmable data-plane pipelines. They are directly relevant, but their model capacity remains bounded by switch pipeline stages, table resources, and limited arithmetic support.The other approach executes high-precision models at the network edge using SmartNICs or servers [11], [51], [52], typically introducing additional latency and providing incomplete visibility into network-wide traffic patterns. Taurus [10] is a closely related architecture-oriented effort that supports per-packet ML through additional data-plane compute and validates the design with FPGA prototyping and ASIC synthesis.

FENIX differs by exploring a complementary point in the design space. It provides a complete co-design methodology for high-throughput DNN inference in the data plane, addressing the throughput gap between heterogeneous chips through workload partitioning and pipelining. Rather than placing the entire DNN inside the switch pipeline, as in Quark and Pegasus, or relying on new data-plane compute primitives, as in Taurus, FENIX demonstrates a deployable Tofino 2–ZU19EG prototype with switch-side feature extraction, FPGA-side DNN execution, and probabilistic rate-controlled offload.By conducting real-world evaluations across multiple traffic analysis tasks, FENIX demonstrates that advanced neural inference capabilities can be integrated into switch-centric architectures while maintaining line-rate performance. This work provides concrete design insights, including resource allocation strategies, inter-chip communication protocols, and task-specific optimizations, that could inform both commercial deployments and future ASIC developments for intelligent network data planes.

X. Conclusion

In this paper, we present FENIX, a hybrid in-network machine learning system designed to enable fast and accurate traffic analysis directly in the data plane. FENIX splits the workload by performing feature extraction on programmable switches and delegating DNN inference to FPGAs, allowing the system to achieve both high throughput and low latency without sacrificing accuracy. To address the bandwidth gap between switches and FPGAs, FENIX introduces a token bucket-based mechanism to efficiently regulate feature transmission. Our implementation and evaluation on real-world traffic show that FENIX delivers microsecond-level inference latency, multi-terabit throughput, and over 90% classification accuracy with minimal hardware overhead. Our results serve as a compelling proof-of-concept for the viability of future intelligent data-plane switches.

References

[1] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, "Kitsune: An ensemble of autoencoders for online network intrusion detection," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.

[2] C. Fu, Q. Li, M. Shen, and K. Xu, "Realtime robust malicious traffic detection via frequency domain analysis," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2021, pp. 3431–3446.

[3] G. Zhou, Z. Liu, C. Fu, Q. Li, and K. Xu, "An efficient design of intelligent network data plane," in *Proc. 32nd USENIX Secur. Symp.*, 2023, pp. 6203–6220.

[4] S. U. Jafri, S. Rao, V. Shrivastav, and M. Tawarmalani, "Leo: Online ML-based traffic classification at multi-terabit line rate," in *Proc. USENIX Symp. Networked Syst. Design Implement. (NSDI)*, 2024, pp. 1573–1591.

[5] D. Barradas, N. Santos, L. Rodrigues, S. Signorello, F. M. V. Ramos, and A. Madeira, "FlowLens: Enabling efficient flow classification for ML-based network security applications," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2021, pp. 1–18.

[6] J. Yan, H. Xu, Z. Liu, Q. Li, K. Xu, M. Xu, and J. Wu, "Brain-on-switch: Towards advanced intelligent network data plane via NN-driven traffic analysis at line-speed," in *Proc. USENIX Symp. Networked Syst. Design Implement. (NSDI)*, 2024, pp. 419–440.

[7] P. Bosshart et al., "p4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[8] *NetFPGA*. NetFPGA. Accessed: Jan. 2025. [Online]. Available: https://netfpga.org/

[9] D. Firestone et al., "Azure accelerated networking: SmartNICs in the public cloud," in *Proc. USENIX Symp. Networked Syst. Design Implement. (NSDI)*, 2018, pp. 51–66.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:09:32 UTC from IEEE Xplore. Restrictions apply.

[10] T. Swamy, A. Rucker, M. Shahbaz, I. Gaur, and K. Olukotun, "Taurus: A data plane architecture for per-packet ML," in *Proc. 27th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Feb. 2022, pp. 1099–1114.

[11] G. Siracusano et al., "Re-architecting traffic analysis with neural network interface cards," in *Proc. 19th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2022, pp. 513–533.

[12] NVIDIA. *NVIDIA ConnectX-7 SmartNICs*. Accessed: Jan. 2025. [Online]. Available: https://www.nvidia.cn/networking/ethernet-adapters/

[13] Intel. *Intel TofinoT 2*. Accessed: Jan. 2025. [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/218648/intel-tofino-2-12-8-tbps-20-stage-4-pipelines/specifications.html

[14] Intel. *Intel Intelligent Fabric Processors*. Accessed: Jan. 2025. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors.html

[15] S. Gupta et al., "Loom: Flexible and efficient NIC packet scheduling," in *Proc. USENIX Symp. Networked Syst. Design Implement. (NSDI)*, 2019, pp. 73–88.

[16] X. Gao et al., "FENIX: Enabling in-network DNN inference with FPGA-enhanced programmable switches," 2025, *arXiv:2507.14891*.

[17] C. Zeng et al., "Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing," in *Proc. 19th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2022, pp. 1345–1358.

[18] W. Feng et al., "f3: Fast and flexible network telemetry with an FPGA coprocessor," *Proc. ACM Netw.*, vol. 2, pp. 1–22, Dec. 2024.

[19] Advanced Micro Devices (AMD). *AMD ZynqT UltraScale+T MPSoC*. Accessed: Jan. 2025. [Online]. Available: https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-ultrascale-plus-mpsoc.html

[20] G. Xie, Q. Li, Y. Dong, G. Duan, Y. Jiang, and J. Duan, "Mousika: Enable general in-network intelligence in programmable switches by knowledge distillation," in *Proc. IEEE Conf. Comput. Commun.*, May 2022, pp. 1938–1947.

[21] J.-H. Lee and K. Singh, "SwitchTree: In-network computing and traffic analyses with random forests," *Neural Comput. Appl.*, vol. 37, no. 28, pp. 1–12, Oct. 2025.

[22] C. Zheng et al., "IIsy: Practical in-network classification," 2022, *arXiv:2205.08243*.

[23] S. Kim, S. M. M. Mirnajafizadeh, B. Kim, R. Jang, and D. Nyang, "SketchFeature: High-quality per-flow feature extractor towards security-aware data plane," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2025, pp. 1–17.

[24] B. Spang and N. McKeown, "On estimating the number of flows," in *Proc. Stanford Workshop Buffer Sizing*, 2019, pp. 1–4.

[25] AMD. *Vitis High-Level Synthesis User Guide (UG1399)*. Accessed: Jan. 2025. [Online]. Available: https://docs.amd.com/r/2024.1-English/ug1399-vitis-hls/Introduction

[26] *Vitis AI 3.5 Documentation*. Accessed: Jan. 2025. [Online]. Available: https://xilinx.github.io/Vitis-AI/3.5/html/index.html

[27] G. Draper-Gil, A. H. Lashkari, M. S. I. Mamun, and A. A. Ghorbani, "Characterization of encrypted and VPN traffic using time-related features," in *Proc. 2nd Int. Conf. Inf. Syst. Secur. Privacy*, 2016, pp. 407–414.

[28] W. Wang, M. Zhu, X. Zeng, X. Ye, and Y. Sheng, "Malware traffic classification using convolutional neural network for representation learning," in *Proc. Int. Conf. Inf. Netw. (ICOIN)*, Da Nang, Vietnam, Jan. 2017, pp. 712–717.

[29] *DPDK Project. DPDK—The Open Source Data Plane Development Kit Accelerating Network Performance*. Accessed: Jan. 2025. [Online]. Available: https://www.dpdk.org/

[30] XGBoost Developers. *Introduction to XGBoost Python Package*. Accessed: Jan. 2025. [Online]. Available: https://xgboost.readthedocs.io/en/stable/python/python_intro.html

[31] Intel. *Product Change Notification: Tofino 2*. Accessed: Jan. 2025. [Online]. Available: https://cdrdv2-public.intel.com/827577/PCN827577-00.pdf

[32] M. Yang et al., "Using trio: Juniper networks' programmable chipset–for emerging in-network applications," in *Proc. ACM SIGCOMM Conf.*, Aug. 2022, pp. 633–648.

[33] B. Inc. *BCM78800: StrataXGS Trident 5 Ethernet Switch Series*. Accessed: Jan. 2025. [Online]. Available: https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm78800

[34] Y. Feng et al., "Enabling in-situ programmability in network data plane: From architecture to language," in *Proc. 19th USENIX Symp. Networked Syst. Design Implement.*, 2022, pp. 635–649.

[35] J. Xing et al., "Runtime programmable switches," in *Proc. 19th USENIX Symp. Networked Syst. Design Implement.*, 2022, pp. 651–665.

[36] Y. Feng et al., "Empower programmable pipeline for advanced stateful packet processing," in *Proc. USENIX Symp. Networked Syst. Design Implement. (NSDI)*, 2024, pp. 491–508.

[37] S. Chole et al., "dRMT: Disaggregated programmable switching," in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2017, pp. 1–14.

[38] T. Wang, X. Yang, G. Antichi, A. Sivaraman, and A. Panda, "Isolation mechanisms for high-speedpacket-processing pipelines," in *Proc. USENIX Symp. Networked Syst. Design Implement. (NSDI)*, 2022, pp. 1289–1305.

[39] T. Swamy, A. Zulfiqar, L. Nardi, M. Shahbaz, and K. Olukotun, "Homunculus: Auto-generating efficient data-plane ML pipelines for datacenter networks," in *Proc. 28th ACM Int. Conf. Architect. Support Program. Lang. Oper. Syst.*, vol. 3, 2023, pp. 329–342.

[40] W. Xu et al., "ClickINC: In-network computing as a service in heterogeneous programmable data-center networks," in *Proc. ACM SIGCOMM Conf.*, Sep. 2023, pp. 798–815.

[41] J. Romeiras Amado, F. Pereira, D. Pissarra, S. Signorello, M. Correia, and F. M. V. Ramos, "Peregrine: ML-based malicious traffic detection for terabit networks," 2024, *arXiv:2403.18788*.

[42] Y. Qing et al., "Low-quality training data only? A robust framework for detecting encrypted malicious network traffic," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2024, pp. 1–18.

[43] V. Rimmer, D. Preuveneers, M. Juarez, T. V. Goethem, and W. Joosen, "Automated website fingerprinting through deep learning," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.

[44] X. Deng et al., "Robust multi-tab website fingerprinting attacks in the wild," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2023, pp. 1005–1022.

[45] M. Shen, Y. Liu, L. Zhu, X. Du, and J. Hu, "Fine-grained webpage fingerprinting using only packet length information of encrypted traffic," *IEEE Trans. Inf. Forensics Security*, vol. 16, pp. 2046–2059, 2020.

[46] K. Al-Naami et al., "Adaptive encrypted traffic fingerprinting with bi-directional dependence," in *Proc. 32nd Annu. Conf. Comput. Secur. Appl.*, Dec. 2016, pp. 177–188.

[47] M. Shen, J. Zhang, L. Zhu, K. Xu, and X. Du, "Accurate decentralized application identification via encrypted traffic analysis using graph neural networks," *IEEE Trans. Inf. Forensics Security*, vol. 16, pp. 2367–2380, 2021.

[48] X. Lin, G. Xiong, G. Gou, Z. Li, J. Shi, and J. Yu, "ET-BERT: A contextualized datagram representation with pre-training transformers for encrypted traffic classification," in *Proc. ACM Web Conf.*, Apr. 2022, pp. 633–642.

[49] R. Zhao et al., "Yet another traffic classifier: A masked autoencoder based traffic transformer with multi-level flow representation," in *Proc. AAAI Conf. Artif. Intell.*, 2023, pp. 5420–5427.

[50] G. Zhou, X. Guo, Z. Liu, T. Li, Q. Li, and K. Xu, "TrafficFormer: An efficient pre-trained model for traffic data," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2025, pp. 1844–1860.

[51] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat, "SENIC: Scalable NIC for end-host rate limiting," in *Proc. USENIX Symp. Networked Syst. Design Implement. (NSDI)*, 2014, pp. 475–488.

[52] G. Siracusano et al., "Running neural networks on the NIC," 2020, *arXiv:2009.02353*.

[53] M. T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, and D. Wentzlaff, "Enabling programmable transport protocols high-speed nics," in *Proc. USENIX Symp. Networked Syst. Design Implement. (NSDI)*, 2020, pp. 93–109.

[54] Z. Wang, Z. Liu, X. Wang, S. Fu, and K. Xu, "DIP: Unifying network layer innovations using shared L3 core functions," in *Proc. 21st ACM Workshop Hot Topics Netw.*, Nov. 2022, pp. 60–67.

[55] A. G. Alcoz, M. Strohmeier, V. Lenders, and L. Vanbever, "Aggregate-based congestion control for pulse-wave DDoS defense," in *Proc. ACM SIGCOMM Conf.*, New York, NY, USA, 2022, pp. 693–706.

[56] Z. Liu et al., "Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric DDoS attacks with programmable switches," in *Proc. USENIX Secur. Symp.*, 2021, pp. 3829–3846.

[57] J. Xing, Q. Kang, and A. Chen, "NetWarden: Mitigating network covert channels while preserving performance," in *Proc. USENIX Secur. Symp. (USENIX Secur.)*, 2020, pp. 2039–2056.

[58] G. Li et al., "Enabling performant, flexible and cost-efficient DDoS defense with programmable switches," *IEEE/ACM Trans. Netw.*, vol. 29, no. 4, pp. 1509–1526, Aug. 2021.

[59] Y. Zhang et al., "Pegasus: A universal framework for scalable deep learning inference on the dataplane," in *Proc. ACM SIGCOMM Conf.*, Sep. 2025, pp. 692–706.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:09:32 UTC from IEEE Xplore. Restrictions apply.

[60] A. T.-J. Akem, B. Bütün, M. Gucciardo, and M. Fiore, "Jewel: Resource-efficient joint packet and flow level inference in programmable switches," in *Proc. IEEE Conf. Comput. Commun.*, May 2024, pp. 1631–1640.
[61] C. Busse-Grawitz, R. Meier, A. Dietmüller, T. Bühler, and L. Vanbever, "PForest: In-network inference with random forests," 2019, *arXiv:1909.05680*.
[62] A. T.-J. Akem, M. Gucciardo, and M. Fiore, "Flowrest: Practical flow-level inference in programmable switches with random forests," in *Proc. IEEE Conf. Comput. Commun.*, May 2023, pp. 1–10.
[63] H. Huang et al., "Re-architecting buffer management in lossless Ethernet," *IEEE/ACM Trans. Netw.*, vol. 32, no. 6, pp. 4749–4764, Dec. 2024.
[64] X. Du et al., "PRED: Performance-oriented random early detection for consistently stable performance in datacenters," in *Proc. USENIX Symp. Networked Syst. Design Implement. (NSDI)*, 2025, pp. 1–20.
[65] H. Huang, K. Xu, T. Li, Z. Liu, X. Du, and X. Gao, "DiffECN: Differential ECN marking for datacenter networks," *IEEE Trans. Netw.*, vol. 33, no. 1, pp. 210–225, Feb. 2025.
[66] M. B. Anwer, M. Motiwala, M. M. BTariq, and N. Feamster, "Switchblade: A platform for rapid deployment of network protocols," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2010, pp. 183–194.
[67] A. Sivaraman et al., "Programmable packet scheduling at line rate," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 44–57.
[68] T. Li et al., "TACK: Improving wireless transport performance by taming acknowledgments," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Architectures, Protocols Comput. Commun.*, Jul. 2020, pp. 15–30.
[69] M. Scazzariello, T. Caiazzi, H. Ghasemirahni, T. Barbette, D. Kostić, and M. Chiesa, "A high-speed stateful packet processing approach for tbps programmable switches," in *Proc. USENIX Symp. Networked Syst. Design Implement. (NSDI)*, 2023, pp. 1237–1255.
[70] APS Networks GmbH. (2021). *APS6120Q Advanced Programmable Switch*. Accessed: Oct. 2025. [Online]. Available: https://www.aps-networks.com/wp-content/uploads/2021/06/210616_APS6120Q_preliminary.pdf
[71] Cisco Systems. *Cisco N9300 Series Smart Switches*. Accessed: Oct. 2025. [Online]. Available: https://www.cisco.com/site/us/en/products/networking/cloud-networking-switches/9300-series-smart-switches/index.html
[72] M. Zhang et al., "Quark: Implementing convolutional neural networks entirely on programmable data plane," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, May 2025, pp. 1–10.

Xiangyu Gao (Graduate Student Member, IEEE) received the B.S. degree from Taishan College, Shandong University, Qingdao, China, in 2021. He is currently pursuing the Ph.D. degree with Tsinghua University, Beijing, China. His research interests include data center networking and programmable data plane.

Tong Li (Member, IEEE) received the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, China, in 2017. He was a Chief Engineer with Huawei in 2022. He is currently a Full Professor with Renmin University of China. His research interests include networking, distributed systems, and big data.

Yinchao Zhang (Member, IEEE) is currently pursuing the Ph.D. degree with the Department of Computer Science and Technology, Tsinghua University, Beijing, China. His primary research interests include intelligent programmable switch architectures.

Ziqiang Wang received the Ph.D. degree from Southeast University. He is currently a Post-Doctoral Researcher with the Department of Computer Science and Technology, Tsinghua University. His research interests include network architecture and security.

Xiangsheng Zeng received the B.S. degree from Huazhong University of Science and Technology, Wuhan, China, in 2026. His research interests include network traffic analysis and artificial intelligence for networking.

Liang Zhu received the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 2017. He is currently with Zhongguancun Laboratory, Beijing, China. His research interests include Internet routing security and intelligent programmable switch architectures.

Su Yao (Senior Member, IEEE) received the Ph.D. degree from Beijing Jiaotong University, Beijing, China. He is currently an Associate Professor with Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University. He has published more than 70 articles in leading networking venues and journals. His research interests include future Internet architecture and protocols, edge computing, and network security. He has received Best Paper Awards at IEEE ICCC and SecureComm. He serves as an Associate Editor.

Ke Xu (Fellow, IEEE) received the Ph.D. degree from Tsinghua University, Beijing, China. He is currently a Full Professor with the Department of Computer Science and Technology, Tsinghua University. He has published more than 200 technical articles in the research areas of next-generation internet, blockchain systems, and network security. He won the IWQoS 24 Best Paper Award and the Distinguished Paper Award at USENIX Security 23/24 and NDSS 25.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:09:32 UTC from IEEE Xplore. Restrictions apply.