PLUTO: A Robust LDoS Attack Defense System Executing at Line Speed

Dan Tang, Boru Liu, Keqin Li, *Fellow, IEEE*, Sheng Xiao, Wei Liang, and Jiliang Zhang

***Abstract*—The Low-Rate Denial of Service (LDoS) attack poses a significant threat to Internet services. Exploiting vulnerabilities in adaptive mechanisms embedded within network protocols, LDoS attacks are covert and exhibit legal behavior, making defense challenging. Existing LDoS attack solutions cannot perform real-time LDoS attack defense at line speed. With the emergence of P4, users can program the per-packet processing logic of the P4 switch, which offers us the chance to propose PLUTO, the first data plane-aware LDoS attack defense system built upon the P4 switch, possessing line-speed execution capacity. To meet the resource constraints of the P4 switch, we propose the time window-based pre-inference strategy to detect LDoS attacks and the time-limited per-flow state management to filter the LDoS attack flows. For the practical deployment, we develop the P4 Function Tool to extend the P4 primitives for more function operations. We also adopt an encoding-based mapping method to deploy the pre-inference model. Furthermore, we develop the async-updated hash table for quickly filtering LDoS attack flows. Compared with the baseline, PLUTO reduces the equal error rate (EER) by 27.96% and the average mitigation response time by 12.749 s, increasing the AUC by 1.83%, the F1 Score by 7.27%, and the Recall by 9.58%.**

***Index Terms*—Attack defense, data plane-aware, LDoS attacks, line-speed execution, P4.**

I. Introduction

NOWADAYS, the Internet is extensively utilized for real-world communication, making its security concerns critically important. Denial of service (DoS), one of the most common cyber attacks, remains a significant threat to Internet services. While traditional DoS attacks exploit brute force (e.g., link flooding) to continuously exhaust network resources, low-rate denial of service (LDoS) attacks send intermittent burst traffic with much smaller overheads. In particular, the LDoS attacks mainly exploit vulnerabilities in the adaptive mechanisms embedded within network protocols (e.g., congestion control mechanism in TCP and request-response mechanism in HTTP, etc.), ultimately achieving malicious occupation of network resources. In 2009, LDoS attacks forced numerous websites in Iran to shut down [1]. On May 13th, 2022, the crucial government and institution sites in Italy suffered from LDoS attacks, resulting in a website outage for one hour at least [2]. In particular, Italy CERT has highlighted that LDoS attacks can evade traditional DoS attack solutions and pose significant challenges to defense.

Concretely, the difficulty in defending against LDoS attacks arises from their three characteristics:

1) *Legal behavior:* Instead of brute force, LDoS attacks exploit negative feedback from the adaptive mechanisms of network protocols, legally seizing network resources.
2) *Concealment:* Since the burst traffic sent by LDoS attacks is transient, it can be easily confused with the short-term burst traffic generated by benign applications [3]. Additionally, the average rate of LDoS attack traffic is low, which makes it can evade long-term detection.
3) *Invisible:* LDoS attacks can directly interfere with the adaptive mechanism, i.e., the congestion control mechanism in TCP, through the end-to-end path. In this scenario, the attack traffic is invisible to the victim hosts. As a result, LDoS attacks can evade detection on the victim side.

To feasibly detect LDoS attacks, existing solutions adopt in-network traffic analysis through traffic mirroring or network function virtualization (NFV). In particular, NFV mainly employs the software-defined networking (SDN) [8], [9] or the software-defined middlebox (SDM). Table I reviews and compares these solutions.

For the solutions utilizing traffic mirroring or SDN (e.g., Kitsune [4], HGB-FP [5], and P&F [6]), they cannot achieve line-speed execution, thus they are unable to offer real-time detection and timely mitigation within high-throughput networks. Besides, the SDM-based solutions, e.g., Whisper, can execute at

TABLE I
Comparing the Existing LDoS Attack Solutions

Category	Instance	Acceptable Accuracy	Line-Speed Execution	Mitigation Capability	Timely Mitigation Response
Traffic Mirroring	Kitsune [4]	✔	✗	✗	-
SDN-based	HGB-FP [5]	✔	✗	✔	✗
	P&F [6]	✔	✗	✔	✗
SDM-based	Whisper [7]	✔	✔	✗	-
P4-based	PLUTO	✔	✔	✔	✔

Received 27 April 2023; revised 5 October 2024; accepted 19 December 2024. Date of publication 25 December 2024; date of current version 15 May 2025. This work was supported in part by the National Natural Science Foundation of China under Grant 62472153 and in part by the Natural Science Foundation General Project of Chongqing under Grant CSTB2022NSCQ-MSX1378. *(Corresponding author: Jiliang Zhang.)*

Dan Tang is with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410012, China, and also with the Research Institute of Hunan University in Chongqing, Chongqing 401120, China (e-mail: Dtang@hnu.edu.cn).

Boru Liu and Sheng Xiao are with the College of Computer Science and Electronic Engineering (CSEE), Hunan University (HNU), Changsha 410012, China (e-mail: liuboru@hnu.edu.cn; xiaosheng@hnu.edu.cn).

Keqin Li is with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA (e-mail: lik@newpaltz.edu).

Wei Liang is with the School of Computer Science and Engineering, Hunan University of Science and Technology (HNUST), Xiangtan 411199, China (e-mail: wliang@hnust.edu.cn).

Jiliang Zhang is with the College of Semiconductors (College of Integrated Circuits), Hunan University, Changsha 410012, China (e-mail: zhangjiliang@hnu.edu.cn).

Digital Object Identifier 10.1109/TDSC.2024.3522104

line speed by taking advantage of Intel data plane development kits (DPDK). However, limited by the isolation effect from virtualization, they cannot directly interfere with the ongoing traffic for LDoS attack mitigation.

Aiming for line-speed detection and mitigation needs, the P4[1] switch emerged in the context of the programmable data plane (PDP). The P4 switch exhibits three significant advantages: (**i**) *Programmable:* The P4 switch supports customizing data plane-aware network functions, enabling the deployment of LDoS attack solutions. (**ii**) *Per-packet Processing Mode.* The P4 switch can apply user-defined processing logic to each packet, which provides fine-grained detection and fast mitigation response against LDoS attacks. (**iii**) *Line-speed Execution.* The LDoS attack solutions can be executed at line speed in the P4 switch. In particular, Intel Tofino 1 supports line-speed execution up to 6.5 Tbps.

To this end, we propose PLUTO, the first LDoS attack defense system (ADS) built upon the P4 switch, exhibiting line-speed execution capacity.

Data Plane-aware Design: To achieve resource friendliness with the P4 switch, we design the LDoS ADS by proposing:

i) time window-based pre-inference strategy,
ii) time-limited per-flow state management (TPSM).

Using the time window-based pre-inference strategy, we analyze the features of aggregate flow in time window units to determine whether the network is experiencing LDoS attacks. Note that, we only maintain one group of states relevant to the aggregate flow, which is significantly lightweight for the P4 switch. Additionally, we adopt ensemble learning (EL) algorithms to train the pre-inference model and extract both time domain and time-frequency domain features from the aggregate flow, achieving robust pre-inference.

Furthermore, if and only if the pre-inference result indicates LDoS attacks have occurred, we enable the TPSM to handle all arrival flows for the Flow-based Attacker Filtering. In particular, within the concept of TPSM, we limit the activation time for per-flow state management, reducing the flow scale that the P4 switch handles. This allows us to practically conduct the Flow-based Attacker Filtering under the resource constraints of the P4 switch. In addition, based on the outlier behavior exhibited by LDoS attacks, we configure a prior rule for per-flow verification, filtering LDoS attack flows quickly.

Deployment Challenge: To actually deploy our LDoS ADS on the P4 switch, we implement the following three modules: (i) P4 Function Tool, (ii) Pre-inference Model Mapping, and (iii) Flow-based Attacker Filtering.

To compute features in the P4 switch, we develop the P4 Function Tool which extends P4 to support more function operations. In particular, the P4 Function Tool is a generic module for diverse P4 programming. In its design, we utilize the binary matching (BM) task to establish the function mapping, and we propose the scope reduction to implement function operations in a memory-friendly manner. Notably, we address four challenges when performing scope reduction in the P4 switch, including: (i) computing the most significant bit index, (ii) the variable-length shift, (iii) the precise scaling, and (iv) the modulo operation.

To achieve the Pre-inference Model Mapping in the P4 switch, we utilize tree-based EL algorithms for training. Meanwhile, we adopt an encoding-based mapping method [10], converting tree models to the longest prefix matching (LPM) task and the ternary matching (TM) task. Specifically, we merge multiple tree models into a single model and prune it to fit the resource constraints of the P4 switch.

To conduct the Flow-based Attacker Filtering, we explore the flow scale in real-world traffic, pre-allocating reasonable and acceptable memory for the TPSM. Besides, we propose the deterministic data structure, i.e., the async-update hash table, to apply prior rule-based per-flow verification in per-packet processing mode. Additionally, we adopt the memory-friendly approximate data structure, i.e., blocked bloom filter, to build a blocklist enabled throughout LDoS ADS runtime.

Evaluation: First, we compare the P4 Function Tool to the baseline. Results indicate that the P4 Function Tool significantly decreases the TCAM usage by an average of 94.52% and exhibits more stable and smaller relative errors. Second, we use a real-world topology and real-world traffic datasets to evaluate the detection and mitigation performance of the PLUTO. Compared with the traditional solution, the PLUTO reduces the equal error rate (EER) by 27.96% and the average mitigation response time by 12.749 s, increasing the area under ROC curve (AUC) by 1.83%, the F1 Score by 7.27%, and the Recall by 9.58%.

In conclusion, our paper has four main contributions:

- We present PLUTO, the first LDoS attack defense system built up on the P4 switch, achieving robust detection and mitigation at line speed.
- We propose the time window-based pre-inference strategy and the time-limited per-flow state management, ensuring the PLUTO is resource-friendly for the P4 switch.
- We develop the P4 Function Tool by using the binary matching task and the scope reduction, providing function operations for generic P4 programming.
- We develop async-update hash table, a P4-based data structure, enabling the Flow-based Attacker Filtering in per-packet processing mode.

The remainder of this paper is structured as follows. Section II introduces the threat model of LDoS attacks, concurrently outlining both the background and related works of P4. Section III shows the high-level design of PLUTO. Further design details are presented in both Sections IV and V. Meanwhile, the implementation of PLUTO is provided in Section VI. Additionally, Section VII demonstrates the experimental configurations and results within the evaluation of PLUTO. Section VIII discusses the potential limitation of PLUTO. Lastly, Section IX reviews this paper overall.

II. Background

In this section, we introduce the threat model of LDoS attacks and provide a summary of the background and related works on P4.

A. Threat Model of LDoS Attacks

Our ADS mainly focuses on the LDoS attacks that target the congestion control mechanism in TCP. Note that, our ADS supports both detection and mitigation for LDoS attacks.

On a macro level, LDoS attacks periodically send pulse traffic to cause the network congestion which is reflected in packet loss events, e.g., packet timeouts or 3-duplicate ACKs. After the congestion control mechanism responds to these events, it reduces the congestion windows and even increase the retransmission timeout (RTO) intervals. As a result, the available TCP bandwidth shrinks repeatedly.

The model and damage of LDoS attacks are illustrated in Fig. 1, where the CUBIC [11] is the default congestion avoidance

[1] P4 refers to Programming Protocol-independent Packet Processors

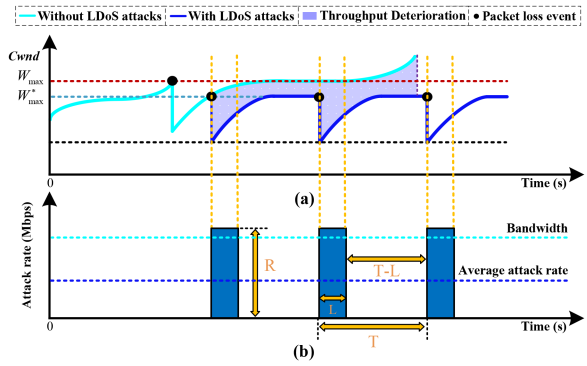


Fig. 1. The Damage (a) and Model (b) of LDoS Attacks under CUBIC.

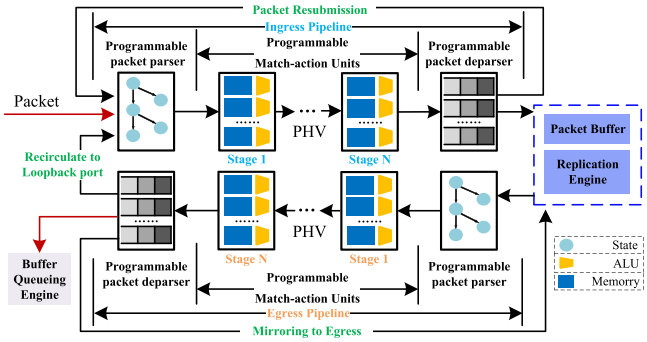


Fig. 2. The Protocol-Independent Switch Architecture (PISA).

algorithm in Linux systems. There are three key parameters relevant to LDoS attacks: attack period (T), burst duration (L), and attack intensity (R). Here, T refers to the time interval between two adjacent pulses, L indicates the duration for sending a pulse, and R represents the bandwidth of the pulse traffic. Consequently, the LDoS attack flow exhibits a periodic burst behavior, i.e., the "null frequency" behavior defined in [12]. During each T, the LDoS attack flow only appears for the time of L and remains silent for the rest of the T.

Additionally, LDoS attacks synchronize the attack period (T) with the RTO (generally T is smaller than the RTO). Since RTO is a dynamically adjusted value, attackers set the attack period (T) to the lower bound of RTO (denoted as minRTO), which is set to 1 s according to RFC 2988 [13]. In addition, to effectively shield the packet transmission, the burst duration (L) must be larger than the round-trip time (RTT), and the attack intensity (R) must reach the bottleneck link bandwidth.

B. The Background of P4

Programming Protocol-independent Packet Processors (P4) was first released in 2014 [14]. It is a hardware description language (HDL) for programming the P4-supported network devices, i.e., the P4 targets, achieving customized per-packet processing. Concretely, the P4 targets involve the P4 switch, NetFPGA, and the SmartNIC, etc.

Domain-specific Architecture: As shown in Fig. 2, P4 targets build upon the protocol-independent switch architecture (PISA), which is based on the reconfigurable match/action table (RMT) architecture. Within the PISA, the per-packet processing logic is divided into the ingress and egress pipelines. Each pipeline is organized into three kinds of P4 programmable blocks: (**i**) *Packet parser.* This block can extract headers from a packet. (**ii**) *Match-action units (MAUs).* These blocks contains the resources, including the memory and ALUs, to execute match/action tasks. They can serially update the packet header vector (PHV) which is formed by both the header and metadata of a packet. Besides, one physical MAU corresponds to one logical MAU Stage in P4. (**iii**) *Packet deparser.* This block can assemble the updated PHV and the original payload as a packet. Commonly, to program a P4 target, there are four principles:

1) Each MAU can only execute once per packet, thus a register in P4 can only be accessed once.
2) Within an MAU, there should be no data dependencies between ALUs, so each field of a PHV can only be modified once in each MAU Stage.
3) Each MAU updates a PHV in match/action mode, thus the per-packet processing logic in P4 is presented as a series of match/action tasks. Concretely, P4 supports the default matching (DM) task, the longest prefix matching (LPM) task, the exact matching (EM) task, the ternary matching (TM) task, and the binary matching (BM) task. Here, the BM task refers to the register index-based matching.
4) Different P4 targets have different P4 target architectures, while a P4 target architecture declares the support range of P4 primitives and the P4 code style.

P4 Switch: The P4 switch is one of the P4 targets executing at line speed. Currently, the Intel tofino switch is the state-of-the-art P4 switch based on ASIC hardware, supporting line-speed execution capacity up to 6.5 Tbps. Its P4 target architecture is the Tofino native architecture (TNA) [15] which declares that the Intel tofino can only support limited P4 primitives, including the addition, the subtraction, and the common bit operations. Besides, Intel Tofino has limited hardware resources and only supports up to 12 MAU stages [16], 120 MB SRAM, and 6.2 MB TCAM per pipeline. Additionally, the software development environment (SDE) of commercial Intel Tofino is not open source.

In contrast, the Behavioral Model v2 (BMv2) switch [17] is an open source P4 switch simulated by the software, its P4 target architecture, i.e., the V1Model, is fully public. Consequently, the BMv2 provides a convenient and economical platform for the P4 development.

C. P4-Related Works

We review existing P4-related works into four aspects.

(1) In-network ML Inference: In-network ML inference provides the foundation for achieving diverse network functions. Associate works can be divided into per-packet and per-flow ML inference. Concretely, Planter [10], Mousika [18], and Taurus [19] all execute per-packet ML inference. Notably, Taurus [19] integrates the existing P4 switch with other hardware to support more ML algorithms. In contrast, both NetBeacon [20] and Flowrest [21] conduct per-flow ML inference. In particular, NetBeacon [20] adopts multi-phase inference for each flow. Additionally, the work in [22] provides an approach for both per-packet and per-flow ML inference. FlowLens [23] presents a compact flow-level feature representation with less information loss. Brain-on-Switch [24] deploys a precision-loss RNN on the P4 switch for per-flow inference.

However, utilizing per-packet features to execute ML inference is lack of robustness [20]. Meanwhile, per-flow ML inference will occupy extensive stateful memory to manage per-flow states continuously.

(2) In-network Security Threat Defense: The P4 switch can defend security threats online at line speed. For the DDoS

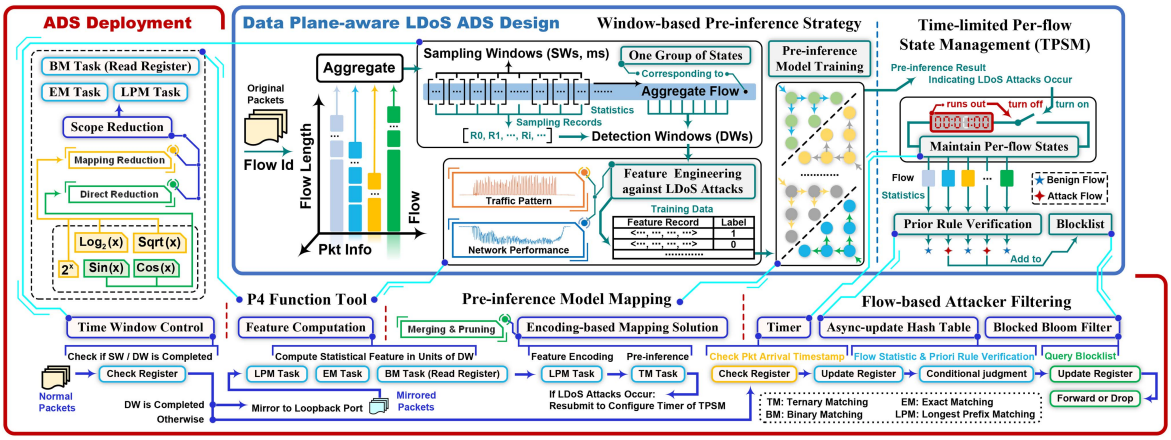


Fig. 3. The Overview of PLUTO.

attacks, Poseidon [25] implements encapsulated primitives for customizing defense strategies. Both the work in [26] and Euclid [27] compute the entropy of IP in the P4 switch for detecting DDoS attacks. Besides, Jaqen [28] prototypes a rule-based DDoS attack defense system with high sensitivity via the P4 switch.

For the variant of DDoS attacks, ACC-Turbo [29] uses a clustering algorithm to defend against pulse-wave DDoS attacks, which exploit different brute-force attack vectors to send periodic pulses. Pulse-wave DDoS attacks are similar to LDoS attacks in periodic pulses, but LDoS attacks do not achieve sufficient traffic surge to fill a cluster, which greatly reduces the accuracy of ACC-Turbo in identifying them, as shown in Fig. 4. Both Ripple [30] and Mew [31] present solutions with efficient distributed resource utilization, defending link-flooding DDoS attacks.

Besides, both P4DDPI [32] and FAPM [33] focus on using P4 to address the leak of DNS privacy. And the work in [34] builds P4-based solution against the abuse of TCP. NetHCF [35] improves existing strategies to counter the IP-spoofing.

(3) Operation Extension for P4: The limited P4 primitives cannot support function operations with real number operands. The work in [36] presents an ideal P4 solutions for computing function operations. But it cannot be realistically deployed on the P4 hardware switch which involves limited MAU Stages. While the work in [37] (denoted as FlexSwitchLib) introduces a practical approach which adopts the longest prefix encoding, achieving function operations in P4 hardware switch. Moreover, FPISA [38] tries to use P4 for conducting float-point real number operations. However, due to the resource isolation of MAUs, FPISA [38] only achieves float-point addition on the the P4 hardware switch, i.e., the Intel Tofino, ultimately.

(4) In-network Measurements: The programmability of the P4 switch enables it to perform the in-network measurement by some crafted designed data structures or algorithms. Both CL-MU [30] and SketchLib [39] implement advanced data structures for in-network measuring and monitoring. Notably, CL-MU [30] can adapt to messy traffic distribution. IMap [40] designs a in-network scanning tool by P4. BeauCoup [41] proposes a parallel data structure, executing multiple queries with one memory access. In addition, Thanos [42] presents an algorithm for maintaining the flow table of the P4 switch. Gallium [43] develops a tool to analyze an optimal strategy for offloading appropriate in-network measurement subprocesses to the P4 switch.

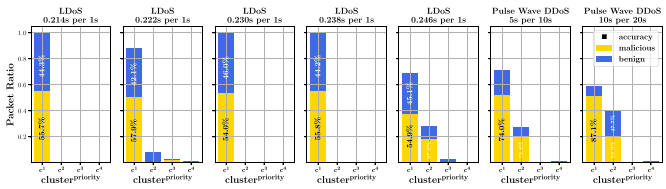


Fig. 4. The Detection Effect of ACC-Turbo on LDoS attacks and Pulse-wave DDoS attacks.

III. Overview of PLUTO

As shown in Fig. 3, our design of PLUTO includes two levels, namely data plane-aware design and ADS deployment.

Data Plane-aware Design: Since there exist extensive arrival flows in a network, e.g., a backbone network, the limited resources in the P4 switch cannot support persistent per-flow state management. To this end, we propose a window-based lightweight pre-inference strategy. It detects the presence of LDoS attacks from a macro perspective without maintaining per-flow states. In detail, on the one hand, we take the time window as the inference unit and adopt the ensemble learning (EL) method to analyze the features of the aggregate flow. Therefore, in the P4 switch, we only need to maintain one group of states corresponding to the aggregate flow. On the other hand, with the advantage of per-packet processing mode in the data plane, we can sample fine-grained statistics about the aggregate flow, resulting in more efficient inferences against LDoS attacks.

Besides, if and only if the pre-inference determines that an LDoS attack has occurred in the network, we enable the time-limited per-flow state management (TPSM) to conduct the Flow-based Attacker Filtering. In the filtering strategy, leveraging the periodic burst behavior typical of LDoS attack flows, we configure a relevant prior rule to verify whether a flow is an LDoS attack flow. The identified flows will be added to the blocklist and prohibited from passing.

ADS Deployment: To practically achieve our data plane-aware design of PLUTO on the P4 switch, we take measures to address the deployment challenges.

To compute statistical features in the P4 switch, we propose P4 Function Tool, a generic P4 module, which extends P4 to support more function operations. In its design, we use the binary matching (BM) task to establish function mapping. Meanwhile, we apply the scope reduction to practically implement the P4 Function Tool in a memory-friendly manner. Notably, the implementation of P4 Function Tool strictly meets the

limitations of the P4 hardware switch, including the limited MAU Stages and the limited P4 primitives. In Section V-A, we will introduce the challenges faced and countermeasures employed during the implementation of the P4 Function Tool.

Besides, we adopt tree-based EL algorithms to build pre-inference model. And we leverage an encoding-based solution to convert the model into the LPM task and the TM task, achieving the pre-inference strategy in the P4 switch. Notably, we merge multiple tree models into a single model and prune it to reduce the usage of MAU Stages and TCAM. The corresponding details will be given in Section V-B.

Additionally, with the time-limited per-flow state management (TPSM), we design a P4-based deterministic data structure, called async-update hash table, to achieve the Flow-based Attacker Filtering in the P4 switch. Meanwhile, we use a resource-friendly approximate data structure, i.e., blocked bloom filter, to implement the blocklist within the P4 switch. In Section V-C, we will clarify the process of Flow-based Attacker Filtering.

IV. Data Plane-Aware Design

In this section, we introduce the details corresponding to our data plane-aware design for PLUTO.

A. Window-Based Pre-Inference

Since the resources in the P4 switch is limited, we cannot utilize it to persistently handle large-scale arrival flows. To this end, we introduce a window-based lightweight pre-inference strategy within our data plane-aware design. Notably, the pre-inference only analyzes the ongoing aggregate flow in the network, thus we only need to maintain one group of states about the aggregate flow in the P4 switch.

We sample the ongoing aggregate flow in units of Sampling Window (SW), where SW is the window on the time scale and its size is denoted as S^{SW}. The sampling record in an SW involves two statistics on aggregate flow, i.e., TCP traffic bytes ($\mathrm{Aggr}_{\mathrm{TB}}$) and overall traffic bytes ($\mathrm{Aggr}_{\mathrm{B}}$).

Furthermore, we perform the feature extraction and the pre-inference in units of Detecting Window (DW). Here, the DW is a sequence containing consecutive S^{DW} sampling records, and S^{DW} represents the size of DW. Therefore, the time consumed by each pre-inference is $(\mathrm{S}^{\mathrm{SW}} \cdot \mathrm{S}^{\mathrm{DW}})$.

Additionally, benefiting from the per-packet processing mode of the P4 switch, we can configure the SWs with tiny time scale to obtain fine-grained statistics on aggregate flow, resulting in more efficient pre-inferences.

B. Time-Limited Per-Flow State Management

When the pre-inference determines that an LDoS attack has occurred within the network, we will activate the TPSM in the P4 switch to filter out LDoS attack flows. In the filtering strategy, based on the outlier behavior exhibited by the LDoS attack flow, i.e., the periodic burst behavior, we configure a relevant prior rule for per-flow verification. This prior rule states that "a flow is identified as an LDoS attack flow if its arrival packets exhibit a periodic burst pattern." The identified flows will be added to the blocklist, and their arrival packets will be dropped directly. In Section V-C, we will introduce how to verify the prior rule in the P4 switch. In contrast, when pre-inference determines that no LDoS attack has occurred on the network, the P4 switch will continue to maintain only one group of states about the aggregate flow.

C. Feature Engineering Against LDoS Attacks

Since the Flow-based Attacker Filtering is enabled based on pre-inference results, the robustness of pre-inference determines the mitigation performance of PLUTO. Therefore, we select features extracted from DW according to two aspects: the network performance and the traffic pattern.

In particular, after extracting all sampling records in a DW, there are two sequences, one for TCP traffic bytes ($\mathrm{Aggr}_{\mathrm{TB}}$) and one for overall traffic bytes ($\mathrm{Aggr}_{\mathrm{B}}$).

Network Performance: LDoS attacks damage the available TCP bandwidth within the victim network. Therefore, we extract time domain statistical features from the sequence of $\mathrm{Aggr}_{\mathrm{TB}}$, reflecting whether the network performance is deteriorated by LDoS attacks.

Traffic Pattern: When LDoS attacks are launched, the "null frequency" behavior exhibited in the attack traffic will interfere with the original traffic pattern composition. Therefore, we convert the sequence of $\mathrm{Aggr}_{\mathrm{B}}$ into the time-frequency domain and utilize the frequency components to indirectly represent the different patterns among the aggregate traffic. Furthermore, we extract statistical features of frequency components to reflect whether the traffic pattern composition is affected by LDoS attacks.

However, the restricted P4 primitives of the P4 switch make feature computation challenging. To this end, we propose the P4 Function Tool in Section V-A for computing features in the P4 switch. And we present the implementation of feature computation in Section VI-B.

V. ADS Deployment

In this section, we design the following three modules to make our data plane-aware design compatible with the P4 switch features: the P4 Function Tool, the Pre-inference Model Mapping, and the Flow-based Attacker Filtering. In each module, we propose solutions to address deployment challenges and issues.

A. P4 Function Tool

To unlock the potential of computing extensive features (e.g., entropy, and variance) with P4, we design the P4 Function Tool, a generic module to extend P4 with more function operations including Logarithm ($\mathrm{Log}_2(\mathrm{x})$), Square Root ($\mathrm{Sqrt}(\mathrm{x})$), Sine ($\mathrm{Sin}(\mathrm{x})$), Cosine($\mathrm{Cos}(\mathrm{x})$), and Exponent ($2^{\mathrm{x}}$). Note that, the P4 Function Tool only utilizes basic P4 primitives supported by the P4 hardware switch, e.g., the Intel Tofino. Meanwhile, it meets the limitation of MAU Stages in the Intel Tofino, i.e., up to 12 MAU Stages.

1) P4 Function Tool Overview: Bit String Format for Real Numbers: We adopt the fixed-point (FP) format to represent real numbers within the function operations.

Given a real number variable, we add a suffix '_fp' to its variable symbol to indicate its FP format. For example, the FP format of the real number x is indicated as $\mathrm{x\_fp}$, and x can be converted to $\mathrm{x\_fp}$ by:

$$\mathrm{x\_fp} = \lfloor \mathrm{x} \ll \mathrm{FRAC\_WIDTH} \rfloor \tag{1}$$

Here, $\mathrm{FRAC\_WIDTH}$ is the bit width of the fractional portion, while the bit width of the integer portion is indicated as $\mathrm{INT\_WIDTH}$. We present the portions of $\mathrm{x\_fp}$ in Fig. 5, and Table II shows their descriptions.

Design Ideas: We conduct the binary matching (BM) task to establish the function mapping. Concretely, assuming that one of the objective functions we need to achieve in P4 is $f(\mathrm{x})$, we

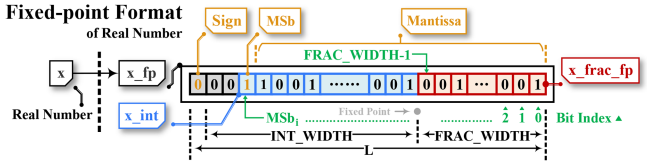


Fig. 5. Fixed-point Format.

TABLE II
DESCRIPTIONS OF THE FP FORMAT PORTIONS

Portion	Description
x_int	It is the integer portion of x in bit string and is indiced by x_fp [FRAC_WIDTH, L − 2].
Sign	It is the sign bit of x_fp.
MSb	It is the most significant bit of x_fp, and its index is $\mathrm{MSb_i}$.
Mantissa	It is the valid portion of x_fp except MSb and is indiced by x_fp $[0, \mathrm{MSb_i} - 1]$.
x_frac	It is the fractional portion of x in bit string and is indiced by x_fp $[0, \mathrm{FRAC\_WIDTH} - 1]$.

```
1  Register ⟨bit ⟨L⟩, bit ⟨16⟩⟩ (NNMS) register_function;
2  RegisterAction ⟨bit ⟨L⟩, bit ⟨16⟩, bit ⟨L⟩⟩ (register_function) fetch_register_function = {
3      void apply (inout bit ⟨L⟩ value, out bit ⟨L⟩ read_value) {
4          read_value = value; } };
```

Fig. 6. P4 Pseudocode for Conducting Binary Matching Task.

Fig. 7. Mapping Reduction.

use the Register and RegisterAction externs to conduct a BM task, as shown in Fig. 6. Each entry of registerfunction stores an indice-value pair with FP format, i.e., $(\mathrm{x\_fp}, f(\mathrm{x})\_\mathrm{fp})$:

$$f(\mathrm{x})\_\mathrm{fp} = \mathrm{fetch\_register\_function}[\mathrm{x\_fp}] \tag{2}$$

Thus we can take x_fp as the indice and execute the RegisterAction, i.e., fetchregisterfunction, to obtain $f(\mathrm{x})\_\mathrm{fp}$.

However, since the function domain $\mathrm{D}(f)$ is significantly wide and even includes negative values, we cannot directly use $\mathrm{D}(f)$ as the matching scope (MS) of BM task. To this end, We propose the scope reduction to investigate a non-negative narrow matching scope (NNMS) to conduct the BM task. There are two categories of scope reduction: mapping reduction and direct reduction.

Mapping Reduction: We denote the mapping reduction process (MRP) of $f(\mathrm{x})$ as:

$$\mathrm{MRP}(f(\mathrm{x})) = \mathrm{MR}\left(\mathrm{portion_{str}}, f'(\mathrm{x_p}),\ \mathrm{g}\right) \tag{3}$$

In this context, $\mathrm{portion_{str}}$ indicates a specific portion of the given x_fp, and we denote the potion's bit string as $\mathrm{x_p}$. Notably, the numeric value of $\mathrm{x_p}$ is normalized. In addition, $f'(\mathrm{x_p})$ is a constructed function with respect to $\mathrm{x_p}$, and g is a mapping which satisfies the following relation:

$$f(\mathrm{x})\_\mathrm{fp} = g(f'(\mathrm{x_p})\_\mathrm{fp}) \tag{4}$$

As shown in Fig. 7, there are three steps in $\mathrm{MRP}(f(\mathrm{x}))$. We utilize $\mathrm{x_p}$_fp as the register indice, thus the NNMS of current BM task is $[0, 1)$. Additionally, we replace the original objective function $f(\mathrm{x})$ with $f'(\mathrm{x_p})$ for the BM task. As a result, we first derive $f'(\mathrm{x_p})\_\mathrm{fp}$ by the BM task and further employ the mapping g to indirectly obtain $f(\mathrm{x})\_\mathrm{fp}$.

Direct Reduction: We perform direct reduction on $f(\mathrm{x})$ when it exhibits properties of parity, periodicity, and symmetry. The

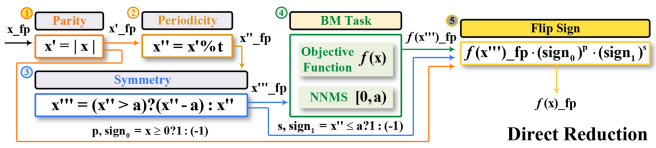


Fig. 8. Direct Reduction.

direct reduction process of $f(\mathrm{x})$ is denoted as:

$$\mathrm{DRP}(f(\mathrm{x})) = \mathrm{DR}\left(\mathrm{p}, \mathrm{t}, \mathrm{a}, \mathrm{s}\right). \tag{5}$$

Here, symbol t represents the period of $f(\mathrm{x})$. Symbol a indicates that the symmetry axis of $f(\mathrm{x})$ is located at $\mathrm{x} = \mathrm{a}$. Symbol p is a boolean indicating the parity of $f(\mathrm{x})$, where $'0'$ corresponds to an even function, and $'1'$ corresponds to an odd function. Symbol s is another boolean indicating the symmetry type of $f(\mathrm{x})$, where $'0'$ represents axial symmetry, and $'1'$ represents center symmetry.

There are five steps in $\mathrm{DRP}(f(\mathrm{x}))$, as illustrated in Fig. 8. We convert x_fp to x'''_fp and utilize x'''_fp as the register indice, thus the NNMS of current BM task is $[0, \mathrm{a})$. Consequently, after deciding the sign bit of $f(\mathrm{x})$_fp based on p and s, we directly obtain $f(\mathrm{x})$_fp by the BM task.

Logarithm and Square Root: We perform the mapping reduction on both $\mathrm{Log_2(x)}$ and $\mathrm{sqrt(x)}$ according to:

$$\mathrm{Mantissa^{norm}} = \mathrm{x\_fp}[\mathrm{MSb_i} - 1]/2^1 + \cdots + \mathrm{x\_fp}[0]/2^{\mathrm{MSb_i}} \tag{6}$$

$$\mathrm{Log_2(x\_fp)\_fp} = (\mathrm{MSb_i})\_\mathrm{fp} + [\mathrm{Log_2}(1 + \mathrm{Mantissa^{norm}})]\_\mathrm{fp} \tag{7}$$

$$\mathrm{Sqrt(x\_fp)\_fp} = 2^{\mathrm{MSb_i}/2} \cdot [\mathrm{Sqrt}(1 + \mathrm{Mantissa^{norm}})]\_\mathrm{fp} \tag{8}$$

$$\mathrm{Log_2(x)\_fp} = \mathrm{Log_2(x\_fp)\_fp} - (\mathrm{FRAC\_WIDTH})\_\mathrm{fp} \tag{9}$$

$$\mathrm{Sqrt(x)\_fp} = \mathrm{Sqrt(x\_fp)\_fp} \ll (\mathrm{FRAC\_WIDTH}/2) \tag{10}$$

Thus the MRPs of $\mathrm{Log_2(x)}$ and $\mathrm{sqrt(x)}$ are respectively:

$$\mathrm{MRP(Log_2(x))} = \mathrm{MR}\left('\mathrm{Mantissa'}, \mathrm{Log_2(Mantissa^{norm} + 1)},\ \mathrm{g_0}\right) \tag{11}$$

$$\mathrm{MRP(Sqrt(x))} = \mathrm{MR}\left('\mathrm{Mantissa'}, \mathrm{Sqrt(Mantissa^{norm} + 1)}, \mathrm{g_1}\right) \tag{12}$$

Besides, the two mappings g_0 and g_1 are respectively:

$$g_0(\mathrm{x\_fp}) = \mathrm{x\_fp} + (\mathrm{MSb_i})\_\mathrm{fp} - (\mathrm{FRAC\_WIDTH})\_\mathrm{fp} \tag{13}$$

$$g_1(\mathrm{x\_fp}) = (\mathrm{x\_fp} \ll 2^{(\mathrm{MSb_i} + \mathrm{FRAC\_WIDTH})//2})) \cdot \left(\sqrt{2}\right)^{\mathrm{MSb_i}\&1} \tag{14}$$

Sine and Cosine: We perform the direct reduction on both $\mathrm{Sin(x)}$ and $\mathrm{Cos(x)}$, their DRPs are respectively:

$$\mathrm{DRP(Sin(x))} = \mathrm{DR}\left(1, 2\pi, \pi, 1\right), \mathrm{DRP(Cos(x))} = \mathrm{DR}\left(0, 2\pi, \pi, 0\right) \tag{15}$$

```
action set_msb_i (bit ⟨L⟩ p0) { ig_md.msb_i = p0; }
table cal_msb_i_table { key={ ig_md.x_fp:lpm; } actions={ set_msb_i; } }
apply {cal_msb_i_table.apply(); // 1st Stage }
```

Fig. 9. P4 Pseudocode for Computing $\mathrm{MSb_i}$.

TABLE III
LONGEST PREFIX MATCHING TABLE FOR COMPUTING $\mathrm{MSb_i}$

Number	Match Value	Prefix Length	Action	Params (p0)
0	0	0	set_msb_i(bit⟨L⟩ p0)	$(\mathrm{L}-0-1)\_\mathrm{fp}$
...	...	...	...	...
i	0	0	set_msb_i(bit⟨L⟩ p0)	$(\mathrm{L}-\mathrm{i}-1)\_\mathrm{fp}$
...	...	...	...	...
$\mathrm{L}-1$	0	$\mathrm{L}-1$	set_msb_i(bit⟨L⟩ p0)	$[\mathrm{L}-(\mathrm{L}-1)-1]\_\mathrm{fp}$

Exponent: We perform the mapping reduction on 2^{x} by:

$$2^{\mathrm{x}}\_\mathrm{fp} = \begin{cases} (2^{|\mathrm{x}|\_\mathrm{frac}})\_\mathrm{fp} \ll |\mathrm{x}|\_\mathrm{int}, \ \mathrm{x} \geq 0 \\ (1/2^{|\mathrm{x}|\_\mathrm{frac}})\_\mathrm{fp} \gg |\mathrm{x}|\_\mathrm{int}, \ \mathrm{x} < 0 \end{cases} \tag{16}$$

Thus the MRP of 2^{x} is:

$$\mathrm{MRP}(2^{\mathrm{x}}) = \begin{cases} \mathrm{MR}\,('\mathrm{x\_frac}', 2^{|\mathrm{x}|\_\mathrm{frac}}, g_2), \ \mathrm{x} \geq 0 \\ \mathrm{MR}\,('\mathrm{x\_frac}', 1/2^{|\mathrm{x}|\_\mathrm{frac}}, g_2), \ \mathrm{x} < 0 \end{cases} \tag{17}$$

Here, the mapping g_2 is:

$$g_2(\mathrm{x\_fp}) = (\mathrm{x} \geq 0) \ ? \ (\mathrm{x\_fp} \ll |\mathrm{x}|\_\mathrm{int}) : (\mathrm{x\_fp} \gg |\mathrm{x}|\_\mathrm{int}) \tag{18}$$

Overall: We apply the MRPs and DRPs shown in (11), (12), (15), and (17), achieving the function operations (i.e., $\mathrm{Log_2(x)}$, $\mathrm{Sqrt(x)}$, $\mathrm{Sin(x)}$, $\mathrm{Cos(x)}$, and 2^{x}) in P4.

2) Challenge: However, due to the limitations of P4 primitives and MAU Stages, there exist four challenges when we practically applying the MRPs and DRPs: (i) the computation of $\mathrm{MSb_i}$, (ii) the variable-length shift, (iii) the precise scaling, (iv) the modulo operation. To overcome these challenges, we propose the following countermeasures.

Computation of $\mathbf{MSb_i}$*:* Existing methods for computing $\mathrm{MSb_i}$, e.g., iteration, bisection, or [26], require more than twelve MAU Stages. Instead, we use a longest prefix matching (LPM) task and consume one MAU Stage to achieve the computation of $\mathrm{MSb_i}$ in P4, as shown in Fig. 9.

We pre-install an LPM table, i.e., the cal_msb_i_table, by L LPM entries, as shown in Table III. For $\mathrm{entry_i^{LPM}}$, its match value is set to 0, its prefix length is set to i, and its parameter p0 is set to $(\mathrm{L}-\mathrm{i}-1)\_\mathrm{fp}$. We use the given $\mathrm{x\_fp}$ as the matching key of cal_msb_i_table, and the relevant $\mathrm{MSb_i\_fp}$ is the parameter p0 in the matched LPM entry.

For instance, consider the case where $\mathrm{L}=8$, for the bit string 00010101, it matches to $\mathrm{entry_3^{LPM}}$, thus its $\mathrm{MSb_i\_fp}$ is set to the parameter $\mathrm{p_0}$ of $\mathrm{entry_3^{LPM}}$, where $\mathrm{p_0} = (\mathrm{L}-3-1)\_\mathrm{fp} = (4)\_\mathrm{fp}$.

Variable-length Shift: We employ an exact matching (EM) task and consume one MAU Stage to achieve variable-length shift in P4, as depicted in Fig. 10. We encapsulate L actions and pre-install L EM entries in an exact matching table, i.e., the variable_len_shift_table. For $\mathrm{entry_i^{EM}}$, its match value is set to i and its action is set to shift_i(). We use a variable $\mathrm{shift\_len}$ as the matching key, and the matched action will shift the given $\mathrm{x\_fp}$ by $\mathrm{shift\_len}$.

Precise Scaling: Although the Intel Tofino supports the MathUnit extern for approximate scaling, it uses only the highest four bits of an operand to perform a scaling, resulting in significant errors. To perform a precise scaling in P4, we expand

```
#define LEFT_SHIFT(len) action shift_##len() {ig_md.x_fp = ig_md.x_fp ≪ len;}
#define RIGHT_SHIFT(len) action shift_##len() {ig_md.x_fp = ig_md.x_fp ≫ len;}
#ifdef LEFT_SHIFT
LEFT_SHIFT(0) LEFT_SHIFT(1) ..... LEFT_SHIFT(L − 1)
#else
RIGHT_SHIFT(0) RIGHT_SHIFT(1) ..... RIGHT_SHIFT(L − 1)
table variable_len_shift_table {
    key={ig_md.shift_len:exact;}
    actions={shift_0; shift_1; shift_2; shift_3; shift_4; shift_5; .....}
}
apply {variable_len_shift_table.apply(); // 1st Stage }
```

Fig. 10. P4 Pseudocode for Variable-length Shift.

```
action _2pi_scaling_0() { ig_md.term0 = ig_md.x_fp ≪ 2; ig_md.term1 = ig_md.x_fp ≪ 1;.....
    ig_md.term13 = ig_md.x_fp ≫ 22; ig_md.term14 =ig_md.x_fp ≫ 26; }
action _2pi_scaling_1() { ig_md.term0 = ig_md.term0 + ig_md.term1;...
    ig_md.term12 = ig_md.term12 + ig_md.term13; }
action _2pi_scaling_2() { ig_md.term0 = ig_md.term0 + ig_md.term2;...
    ig_md.term12 = ig_md.term12 + ig_md.term14; }
action _2pi_scaling_3() { ig_md.term0 = ig_md.term0 + ig_md.term4;...
    ig_md.term8 = ig_md.term8 + ig_md.term12; }
action _2pi_scaling_4() { ig_md.term0 = ig_md.term0 + meta.term8; }
apply {_2pi_scaling_0(); _2pi_scaling_1(); _2pi_scaling_2(); // The 1st to 3rd Stages
_2pi_scaling_3(); _2pi_scaling_4() // The 4th to 5th Stages }
```

Fig. 11. P4 Pseudocode for Precise Scaling.

the scaling factor into $\mathrm{MAX\_EXP\_N}$ terms based on the valid binary weights that are equal to 1. Besides, for the power of each term, its absolute value is less than L. We denote a scaling as $\mathrm{Scal}\,(\sigma, \cdot)$, where σ is the scaling factor. Take $\mathrm{Scal}\,(2\pi, \mathrm{op\_fp})$ as an example, when $\mathrm{L}=32$, we have:

$$\begin{aligned} \mathrm{Scal}(2\pi, \mathrm{op\_fp}) &\approx (2^2 + 2^1 + 2^{-2} + \cdots + 2^{-26}) \cdot \mathrm{op\_fp} \\ &\approx (\mathrm{op\_fp} \cdot 2^2) + (\mathrm{x\_fp} \cdot 2^1) + \frac{\mathrm{op\_fp}}{2^2} + \cdots + \frac{\mathrm{op\_fp}}{2^{26}} \end{aligned} \tag{19}$$

In this case, $\mathrm{MAX\_EXP\_N}$ is 15. As shown in Fig. 11, we encapsulate fixed-length shifts and additions into the five actions, and we apply these actions as default matching (DM) tasks, thus we consume five MAU Stages to achieve $\mathrm{Scal}\,(2\pi, \mathrm{x\_fp})$ in P4. Notably, due to the limited 12 MAU Stages, we only achieve the precise scaling with $\mathrm{L}=32$.

Modulo Operation: If the modulo is not a power of 2, we utilize the precise scaling along with an LPM task to achieve the modulo operation in P4. In line with our solution for the precise scaling, we only consider the case where $\mathrm{L}=32$ to achieve modulo operation in P4. We denote a modulo operation as $\mathrm{MOD}\,(\lambda, \cdot)$, where λ is the modulo value. Take $\mathrm{MOD}\,(\lambda, \mathrm{op\_fp})$ as an example, we convert this modulo operation into two phases of scaling:

$$\mathrm{quo} = \lfloor \mathrm{Scal}\,(1/2\pi, \mathrm{op\_fp}) \gg \mathrm{FRAC\_WIDTH} \rfloor \tag{20}$$

$$\mathrm{rem\_fp} = \mathrm{x\_fp}\%(2\pi)\_\mathrm{fp} = \mathrm{op\_fp} - \mathrm{Scal}\,(2\pi, \mathrm{quo\_fp}) \tag{21}$$

However, due to the limited precision of FP format, the scaling in P4 still exhibits tiny errors. In particular, within (20), The modulo operation cannot tolerate any error of quo which seriously skews its result, i.e., $\mathrm{rem\_fp}$.

To obtain the precise result of $\mathrm{rem\_fp}$, we propose a manner with three steps. First, since the error caused by the scaling is inevitable, we instead perform a fuzzy scaling to compute quo within (20), which reduces the usage of MAU Stages in the P4 switch. In the fuzzy scaling, we expand $1/2\pi$ into $\mathrm{EXP\_N}$ terms, where $\mathrm{EXP\_N} < \mathrm{MAX\_EXP\_N}$.

Second, we further correct $\mathrm{rem\_fp}$ by conducting an LPM task. We denote the error of quo as $\Delta\mathrm{q}$, and $\Delta\mathrm{q}$ can be quantified by $\lfloor \mathrm{rem\_fp}/2\pi \rfloor$. Assuming that there exists max $\Delta\mathrm{q}$, we can infer the scope of $\mathrm{rem\_fp}$ is $\mathrm{S_{rem}}$:

$$\mathrm{rem\_fp} \in \mathrm{S_{rem}} = [0, (\mathrm{Max}\,\Delta\mathrm{q} + 1) \cdot (2\pi)\_\mathrm{fp}] \tag{22}$$

```
action correct_rem (bit ⟨L⟩ p1) { ig_md.rem = ig_md.rem − p1; }
table correct_rem_table { key={ig_md.rem:lpm; } actions={ correct_rem; } }
```

Fig. 12. P4 Pseudocode for Correcting Error Remainder.

TABLE IV
THE USAGE OF LPM ENTRIES AND MAU STAGES UNDER DIFFERENT EXP_N

FRAC_WIDTH	12	10	8	MAU Stages
EXP_N	Max Δq / LPM Entries			
2 (Fuzzy Scaling)	1524/19116	6094/76531	24370/306323	2
4 (Fuzzy Scaling)	244/3047	973/12204	3889/48808	3
8 (Fuzzy Scaling)	4/56	13/164	49/576	4

Besides, we can infer the correcting value for a given rem_fp is $h(\text{rem\_fp})$:

$$h(\text{rem\_fp}) = \lfloor \text{rem\_fp}/2\pi \rfloor \cdot (2\pi)\_\text{fp},\ \text{rem\_fp} \in \text{S}_{\text{rem}} \quad (23)$$

Furthermore, since $h(\text{rem\_fp})$ behaves as a staircase function, we can directly transform it into an LPM task. As shown in Fig. 12. Here, rem_fp serves as the matching key, and $h(\text{rem\_fp})$ corresponds to the parameter p0 in the matched LPM entry. We can apply an LPM table, i.e., the correct_rem_table, to obtain the precise result of rem_fp.

Third, as shown in Table IV, we present the usage of both LPM entries and MAU Stages under different EXP_N. Due to the limited TCAM of the Intel Tofino, we take $\text{EXP\_N} = 8$ to perform the fuzzy scaling. As a result, we consume nine MAU Stages to implement the modulo operation in P4.

3) P4 Function Tool Implementation: In this section, we introduce how to implement each function operation in the P4 switch.

Logarithm and Square Root: We consume three MAU Stages to implement $\text{MRP}(\text{Log}_2(\text{x}))$ and ten MAU Stages to implement $\text{MRP}(\text{Sqrt}(\text{x}))$. As depicted in Fig. 7, both $\text{MRP}(\text{Log}_2(\text{x}))$ and $\text{MRP}(\text{Sqrt}(\text{x}))$ involve three steps.

In the first step, we allocate one MAU Stage to apply the cal_msb_i_table shown in Fig. 9. In particular, we can merge the computation of Mantissa into the action, i.e., set_msb_i. Besides, corresponding to Table III, we extend $\text{entry}_\text{i}^{\text{LPM}}$ with a new parameter p1 which is equal to $2^{(\text{L}-\text{i}-1)}$. Additionally, we use another MAU Stage to apply the variable$_\text{l}$en$_\text{s}$hift$_\text{t}$able shown in Fig. 10, aligning the width of Mantissa to FRAC_WIDTH. Notably, we adopt MSb_i as the shift_len.

In the second step, we use one MAU Stage to execute the RegisterAction, i.e., fetch$_\text{r}$egister$_\text{f}$unction shown in Fig. 6.

In the third step, on the one hand, to compute the mapping g_0 as defined in (13), we initially compute a temporary value of $((\text{MSb}_\text{i})\_\text{fp} - (\text{FRAC\_WIDTH})\_\text{fp})$ by the action, i.e., set_msb_i. And subsequently, we incorporate this temporary value with the BM task result within the execution of fetch$_\text{r}$egister$_\text{f}$unction. Therefore, we do not use additional MAU Stages to compute the mapping g_0.

On the other hand, to compute the mapping g_1 described in (14), we use one MAU Stage to apply another variable$_\text{l}$en$_\text{s}$hift$_\text{t}$able, left-shifting the BM task result by MSb_i. Additionally, leveraging the P4 pseudocode as shown in Fig. 11, we consume 5 MAU Stages to apply the precise scaling within the $\text{MRP}(\text{Sqrt}(\text{x}))$, i.e., $\text{Scal}(\sqrt{2}, \cdot)$.

Overall, the P4 pseudocode of both $\text{MRP}(\text{Log}_2(\text{x}))$ and $\text{MRP}(\text{Sqrt}(\text{x}))$ is presented in Fig. 13.

Sine and Cosine: We consume twelve MAU Stages to implement $\text{DRP}(\text{Sin}(\text{x}))$ or $\text{DRP}(\text{Cos}(\text{x}))$ in P4, each of them

```
.....
action set_msb_i (bit ⟨L⟩ p0, bit ⟨L⟩ p1) {
    ig_md.msb_i_fp = p0; ig_md.mantissa = ig_md.x_fp - p1;
    ig_md.tmp = (int ⟨L⟩) p0 − (10 ≪ FRAC_WIDTH); /* Only executed for Log2(x) */ }
table set_msb_i_table {key={ ig_md.x_fp:lpm; } actions={ set_msb_i; } }
// This modified version of fetch_register_function only for Log2(x)
RegisterAction ⟨bit ⟨L⟩, bit ⟨16⟩, bit ⟨L⟩⟩ (NNMS) fetch_register_function = {
    void apply (inout bit ⟨L⟩ value, out bit ⟨L⟩ read_value) { read_value = value + ig_md.tmp; } }
apply {
    set_msb_i_table.apply(); /* 1st Stage */ variable_len_shift_table_0.apply(); /* 2nd Stage */
    // 3rd Stage, bm_res is the result of Log2(x)
    ig_md.bm_res = fetch_register_function.execute(ig_md.mantissa);
    // for Square Root, sqrt_res is the result of Sqrt(x)
    variable_len_shift_table_1.apply(); // 4th Stage
    if (meta.msb_i_fp & (1 ≪ FRAC_WIDTH)) { // when msb_i is an odd number
        // The 5th to 9th Stages
        sqrt2_scaling_0(); sqrt2_scaling_1(); sqrt2_scaling_2(); sqrt2_scaling_3(); sqrt2_scaling_4();
        ig_md.sqrt_res = ig_md.term0 ≫ (FRAC_WIDTH ≫ 1); // 10th Stage
    } else { meta.sqrt_res = ig_md.em_res ≫ (FRAC_WIDTH ≫ 1); } // 10th Stage }
.....
```

Fig. 13. P4 Pseudocode for Logarithm and Square Root.

```
.....
action correct_rem (bit ⟨32⟩ p0, bit ⟨32⟩ p1) {
    ig_md.rem = meta.rem − p0; ig_md.sign = ig_md.sign ^ p1; }
table correct_rem_table { key={ ig_md.rem:lpm; } actions={ correct_rem; } }
// This modified version of fetch_register_function only for Sin(x) and Cos(x)
RegisterAction ⟨bit ⟨L⟩, bit ⟨16⟩, bit ⟨L⟩⟩ (NNMS) fetch_register_function = {
    void apply (inout bit ⟨L⟩ value, out bit ⟨L⟩ read_value) { read_value = value ^ hdr.mirror_h.sign; } }
apply {
    if (ig_md.x_fp < 0) { ig_md.x_fp = −ig_md.x_fp; ig_md.sign = ig_md.sign ^ 1; } // 1st Stage
    // 1st Stage
    if (hdr.mirror_h.isValid()) { ig_md.bm_res = fetch_register_function.execute(hdr.mirror_h.rem_fp); }
    else {
        // The 2nd to 5th Stages
        recip_2pi_scaling_0(); recip_2pi_scaling_1(); recip_2pi_scaling_2(); recip_2pi_scaling_3();
        // The 6th to 10th Stages
        _2pi_scaling_0(); _2pi_scaling_1(); _2pi_scaling_2(); _2pi_scaling_3(); _2pi_scaling_4();
        ig_md.rem_fp = ig_md.x_fp − ig_md.term0; // 11th Stage
        correct_rem_table.apply(); // 12th Stage
        ig_dprsr_md.mirror_type = MIRROR_TYPE_I2E; // 12th Stage
    }
}
.....
```

Fig. 14. P4 Pseudocode for Sine and Cosine.

involves five steps, as shown in Fig. 8. Within their implementation, we merge the second and third steps by defining the following staircase function $H(\text{rem\_fp})$:

$$H(\text{rem\_fp}) = \begin{cases} h(\text{rem\_fp}), & \text{rem\_fp} - h(\text{rem\_fp}) < (\pi)\_\text{fp} \\ h(\text{rem\_fp}) + (\pi)\_\text{fp}, & \text{Others} \end{cases} \quad (24)$$

Within the $\text{DRP}(\text{Sin}(\text{x}))$ or $\text{DRP}(\text{Cos}(\text{x}))$, we substitute $h(\text{rem\_fp})$ described in (23) with $H(\text{rem\_fp})$ to achieve the modulo operation, i.e., $\text{MOD}(2\pi, .)$. As a result, we can apply the correct$_\text{r}$em$_\text{t}$able presented in Fig. 12 to finish both the second and third steps.

However, achieving the initial three steps consumes twelve MAU Stages, thus we use the Mirror extern to generate a mirrored packet. This packet will re-enter the ingress pipeline and consume one MAU Stage to finish remaining two steps. Note that, Mirror extern does not cause delays for the original packet. Overall, the P4 pseudocode of $\text{DRP}(\text{Sin}(\text{x}))$ is demonstrated in Fig. 14.

Exponent: We consume three MAU Stages to implement $\text{DRP}(2^\text{x})$ in P4. Note that, we conduct two distinct BM tasks corresponding to the cases of $\text{x} \geq 0$ and $\text{x} < 0$ respectively. Under each case, we apply the variable$_\text{l}$en$_\text{s}$hift$_\text{t}$able to left-shift or right-shift the BM task result. The P4 pseudocode of $\text{DRP}(2^\text{x})$ is shown in Fig. 15.

4) Overflow: As shown in Table V, we compute the acceptable operand scope of each function operation. We can configure the hyper-parameters including FRAC_WIDTH and L to avoid operand overflow. For instance, when we configure L as 32 and FRAC_WIDTH as 10, P4 Function Tool can accept the max operand is 2,097,151. However, when we increase L to 64, the max operand can reach 9,007,199,254,740,991, which is hard to overflow.

```
.....
#define BM_TASK(i) Register ⟨bit ⟨L⟩, bit ⟨16⟩⟩ (NNMS) register_exp2_##i; \
RegisterAction ⟨bit ⟨L⟩, bit ⟨16⟩, bit ⟨L⟩⟩ (NNMS) fetch_register_exp2_##i = { \
    void apply (inout bit ⟨L⟩ value, out bit ⟨L⟩ read_value) { \
        read_value = value + ig_md.x_fp[ L − 1 : FRAC_WIDTH ]; } }
BM_TASK(0) BM_TASK(1)
apply {
    if (ig_md.x_fp < 0) { ig_md.x_fp = −ig_md.x_fp; ig_md.sign = 1; } // 1st Stage
    if (ig_md.sign == 1) {
        ig_md.bm_res = fetch_register_exp2_0.execute(ig_md.x_fp[ FRAC_WIDTH - 1 : 0 ]); // 2nd Stage
        variable_len_shift_table_0.apply(); // 3rd Stage
    } else {
        ig_md.bm_res =fetch_register_exp2_1.execute(ig_md.x_fp[ FRAC_WIDTH - 1 : 0 ]); // 2nd Stage
        variable_len_shift_table_1.apply(); // 3rd Stage
    }
}
.....
```

Fig. 15. P4 Pseudocode for Exponent.

TABLE V
THE ACCEPTABLE OPERAND SCOPE OF EACH FUNCTION OPERATION

Function Operation	$\mathrm{Log_2(x)}$	$\mathrm{Sqrt(x)}$	$\mathrm{Sin(x)}$	$\mathrm{Cos(x)}$	2^{x}
Acc Op Scope	$(0, \mathrm{V_0}]$	$[0, \mathrm{V_0}]$	$[-\mathrm{V_1}, \mathrm{V_1}]$	$[-\mathrm{V_1}, \mathrm{V_1}]$	$(-\mathrm{FRAC\_WIDTH}, \mathrm{L}-1)$

$^{1}\mathrm{V_0} = (2^{\mathrm{L}-1}-1)/(2^{\mathrm{FRAC\_WIDTH}})$ and $^{2}\mathrm{V_1} = (2^{31}-1)/(2^{\mathrm{FRAC\_WIDTH}})$

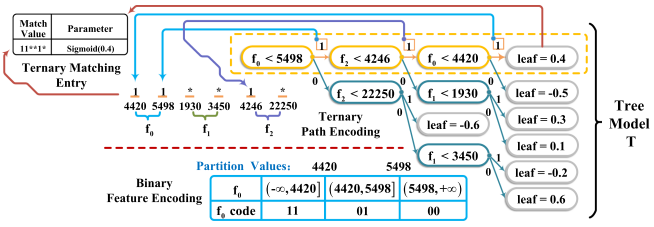


Fig. 16. Encoding-based Mapping.

B. Pre-Inference Model Mapping

To robustly deploy the pre-inference strategy on the P4 switch, we use tree-based EL algorithms to train multiple tree models as the pre-inference model. Specifically, we select three state-of-the-art tree-based EL algorithms, i.e., XGBoost (XGB), RandomForests (RF), and LightGBM (LGBM). The reasons why we choose the tree-based EL algorithms include two points: (i) they do not require normalizing input features; (ii) their inference process is consistent with the match/action task.

However, there are two challenges when actually deploying tree models on the P4 hardware switch, i.e., the Intel Tofino: (i) deploying the tree model by the direct mapping will consume multiple MAU Stages, for example, a tree model with a depth of 4 will consume 4 MAU Stages. (ii) summarizing the inference results of multiple tree models will consume additional MAU Stages, for example, 4 tree models will consumes 2 additional MAU Stages. Therefore, to reduce the consumption of MAU Stages, we adopt an encoding-based mapping method [10] to deploy the tree model on the P4 switch. Besides, we also merge multiple tree models into a single tree model, avoiding the process of summarizing inference results. Additionally, we prune the merged tree model to further decrease the usage of TCAM.

Encoding-based Mapping Method: This method includes binary feature encoding and ternary path encoding. As the tree model T shown in Fig. 16, we first take the $\mathrm{f_0}$ as an example to show the feature encoding process. Since each partition value occupies 1 encoding bit, we use 2 bits for encoding $\mathrm{f_0}$ which has 2 partition values. Concretely, when the $\mathrm{f_0}$ is less than a certain partition value ($\mathrm{parti'}$), the encoding bit of the $\mathrm{parti'}$ is 1, otherwise it is 0. Since the feature encoding process can be abstracted as a staircase function, we use the LPM task to implement the feature encoding in the P4 switch.

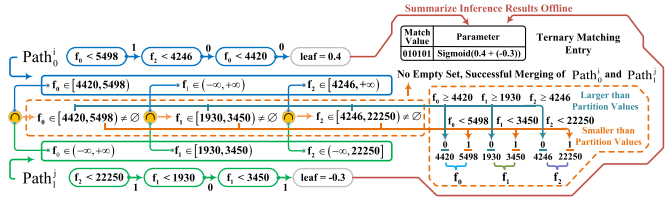


Fig. 17. Merging Two Inference Paths from Different Tree Models.

In addition, as shown in Fig. 16, we take the yellow path as an example to show the path encoding process. Since the tree model T contains 3 features and each feature exhibits 2 partition values, we use a total of 6 bits to encode each path. In the yellow path, $\mathrm{f_0}$ is partitioned twice, and $\mathrm{f_0}$ is smaller than its two partition values, so the $\mathrm{f_0}$ is encoded as 11. $\mathrm{f_1}$ is not used for partitioning, thus $\mathrm{f_1}$ is encoded as $**$. additionally, $\mathrm{f_2}$ is only partitioned by the value of 4246, so the encoding of $\mathrm{f_2}$ is $1*$. In summary, the ternary encoding of the yellow path is $11**1*$.

In the P4 switch, we use a TM task to implement the inference process of the tree model, i.e., the path matching. Each path corresponds to a TM entry. The ternary encoding of a path is the match value in the TM entry, and the inference result is the parameter in the TM entry.

Merging Tree Models: For two tree models $\mathrm{T_0}$ and $\mathrm{T_1}$, their path sets are $\{\mathrm{P_0^i}\}$ and $\{\mathrm{P_1^j}\}$ respectively. Given any i and j, we try to merge $\mathrm{P_0^i}$ and $\mathrm{P_1^j}$. We first indicate the two paths as several feature intervals. Next, we intersect the relevant feature intervals. If and only if all intersection results are not empty sets, $\mathrm{P_0^i}$ and $\mathrm{P_1^j}$ are merged successfully. As shown in Fig. 17, we present a path merging case. Finally, for each merged path, we compute the summarized inference result offline and use it as the parameter of the TM entry.

Pruning: Since the TPSM is initiated only when the pre-inference result is positive, we prune the tree paths with negative inference results.

Overall: We deploy the pre-inference model by only 2 MAU Stages, one for the feature encoding and one for the path matching.

C. Flow-Based Attacker Filtering

When pre-inference determines that an LDoS attack has occurred within the network, we will enable time-limited per-flow state management (TPSM) for the Flow-based Attacker Filtering. Concretely, we need to verify that whether a flow complies with the prior rule, i.e., "a flow is identified as an LDoS attack flow if its arrival packets exhibit a periodic burst pattern." However, when deploying the Flow-based Attacker Filtering on the P4 hardware switch, we need to counter the two issues: (i) How large of a hash array (register) should we pre-allocate for the TPSM? (ii) How can we verify that a flow complies with the prior rule in per-packet processing mode?

Solution for the 1st Issue: We explore the scale of flows in a real backbone network. Note that, consistent with Whisper, we identify the flow by the source IP. We explore the mitigation response time of existing LDoS ADS, we set the activation time of TPSM to 1 minute, which is significantly sufficient for mitigating LDoS attacks. As shown in Fig. 18, we conduct statistics on the real-world traffic collected by the MAWI Working Group [44] on the backbone network from 2022.7.1 to 2022.7.31. The results show that the scale of flows in 1 minute

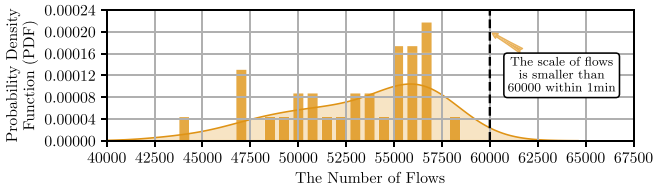


Fig. 18. The Scale of Flows in the Real-world Backbone Network.

Algorithm 1: Flow-Based Attacker Filtering.

Input: Packet: **Pkt**
Output: Action: **Forward** or **Drop**
Initialize IsMaliciousFlow, $\text{bytes}_{\text{burst}}$, and isPeriodic to 0
IsCWCompleted $\Leftarrow$ CW.Update(**Pkt**.arrival_timestamp);
If IsCWCompleted $==1$ **then** $\text{CW}_{\text{id}}^{\text{global}} \Leftarrow \text{CW}_{\text{id}}^{\text{global}} + 1$;
F $\Leftarrow$ hash (**Pkt**.src_IP);
NullCWNum = $\text{CW}_{\text{id}}^{\text{global}} - \text{CW}_{\text{id}}^{\text{flow}}[\text{F}]$;
If NullCWNum ≤ 1 **then** $\text{CW}_{\text{consec}}^{\text{flow}}[\text{F}] \Leftarrow \text{CW}_{\text{consec}}^{\text{flow}}[\text{F}]$ + **Pkt**.len;
Else $\text{bytes}_{\text{burst}} \Leftarrow \text{CW}_{\text{consec}}^{\text{flow}}[\text{F}]$; $\text{CW}_{\text{consec}}^{\text{flow}}[\text{F}] \Leftarrow$ **Pkt**.len;
If NullCWNum == $\text{CW}_{\text{null}}^{\text{flow}}[\text{F}]$ **then** isPeriodic $\Leftarrow 1$;
$\text{CW}_{\text{null}}^{\text{flow}}[\text{F}] \Leftarrow$ NullCWNum;
if $\text{bytes}_{\text{burst}} \geq$ BURST_TH and isPeriodic $==1$ **then**
  IsMaliciousFlow $\Leftarrow 1$; // LDoS Attack Flow
If IsMaliciousFlow $==1$ **then** Blocklist.Add(F);
If F in Blocklist **then** Action $\Leftarrow$ **Drop**;
Else Action $\Leftarrow$ **Forward**;

fluctuates around 60,000. Therefore, while ensuring that the load factor of the hash array does not exceed 50%, the size of the hash array we created is 2^{17}, which is smaller compared with Mew [31] (size is 140000).

Solution for the 2nd Issue: We propose the async-update hash table (AUHT), i.e., a deterministic data structure, serving as a container for executing the prior rule verification. It contains one variable $\text{CW}_{\text{id}}^{\text{global}}$, and three hash arrays with the size of 2^{17}: $\text{CW}_{\text{id}}^{\text{flow}}[\,]$, $\text{bytes}_{\text{consec}}^{\text{flow}}[\,]$, and $\text{num}_{\text{null}}^{\text{flow}}[\,]$.

Within the AUHT, we count the per-flow traffic bytes in units of Counting Window (CW), where CW is the window on the time scale and its size is S^{CW}. In the per-packet processing mode, we cannot uniformly update the states of all flows when a CW is completed. Therefore, we update the states of each flow asynchronously via its arrival packets. Concretely, we use $\text{CW}_{\text{id}}^{\text{global}}$ to store the global CW id. Meanwhile, we use $\text{CW}_{\text{id}}^{\text{flow}}[\,]$ to store the CW id of each flow. Next, for an arrival packet belonging to the flow F, when $\text{CW}_{\text{id}}^{\text{flow}}[\text{F}]$ is not equal to $\text{CW}_{\text{id}}^{\text{global}}$, we update the states of F and synchronize $\text{CW}_{\text{id}}^{\text{flow}}[\text{F}]$ to $\text{CW}_{\text{id}}^{\text{global}}$.

Based on the AUHT, we can verify whether a flow satisfies the prior rule, i.e., "a flow is identified as an LDoS attack flow if its arrival packets exhibit a periodic burst pattern." We utilize $\text{bytes}_{\text{consec}}^{\text{flow}}[\,]$ to store the accumulated traffic bytes under consecutive CWs for each flow. When a packet belonging to the flow F arrives, if $\text{CW}_{\text{id}}^{\text{global}} - \text{CW}_{\text{id}}^{\text{flow}}[\text{F}] \leq 1$, we consider the accumulation of $\text{bytes}_{\text{consec}}^{\text{flow}}[\text{F}]$ is not completed. On the contrary, we take out $\text{bytes}_{\text{consec}}^{\text{flow}}[\text{F}]$ as the burst traffic bytes of F, i.e., $\text{bytes}_{\text{burst}}$. And we reset $\text{bytes}_{\text{consec}}^{\text{flow}}[\text{F}]$.

In addition, we use $\text{num}_{\text{null}}^{\text{flow}}[\,]$ to store the number of consecutive null CWs for each flow. When a packet belonging to the flow F arrives, we check whether $\text{num}_{\text{null}}^{\text{flow}}[\text{F}]$ and $(\text{CW}_{\text{id}}^{\text{global}} - \text{CW}_{\text{id}}^{\text{flow}}[\text{F}])$ are equal. If so, we determine that the flow F is periodic and set a variable, i.e., isPeriodic, to 1. Then we update $\text{num}_{\text{null}}^{\text{flow}}[\text{F}]$ to $(\text{CW}_{\text{id}}^{\text{global}} - \text{CW}_{\text{id}}^{\text{flow}}[\text{F}])$.

To this end, in the per-packet processing mode, when a packet belonging to the flow F arrives, we obtain the variables $\text{bytes}_{\text{burst}}$ and isPeriodic to determine whether F conforms to the prior rule. When $\text{bytes}_{\text{burst}}$ is greater than BURST_TH and isPeriodic is 1, the F is determined to be an LDoS attack flow and added to the blocklist. Overall, the process of Flow-based Attacker Filtering is shown in Algorithm 1.

VI. Implementation

Considering that the software development kit (SDK) of the Intel Tofino series P4 switch is not open source, and PLUTO does not utilize externs (e.g., MathUnit) specifically designed for the Intel Tofino, we implement a prototype of PLUTO on the Behavioral Model v2 (BMv2) switch.

In particular, the implementation of our PLUTO prototype meets the hardware resource constraints of the Intel Tofino 1, i.e., each pipeline only has 12 MAU stages, 120 MB SRAM, and 6.2 MB TCAM. Meanwhile, we present the P4 pseudocode of PLUTO in TNA style, indicating that PLUTO can be deployed on the Intel Tofino.

We take about 1,500 lines of P4_{16} code to implement the switch pipeline as shown in Fig. 19. Besides, we take about 5,000 lines of C++ code to implement a code generator for P4 Function Tool, it generate relevant P4_{16} code for selected function operations. Additionally, we take about 600 lines of Python code to implement both the training and mapping of the pre-inference model.

In the previous Section V-A3, we have presented the implementation of each function operation in P4 Function Tool. In this Section, we introduce the remained implementation including the following aspects: 1) time window; 2) feature computation utilizing P4 Function Tool; 3) Pre-inference Model Mapping; 4) Flow-based Attacker Filtering and blocklist. 5) local CPU control logic. Note that, we configure the FRAC_WIDTH to 10 within the feature computation. The MAU Stage numbers marked in all P4 pseudocodes are consistent with Fig. 19.

A. Time Window

Take the Sampling Window (SW) as an example, as shown in Fig. 20, we utilize a register with one entry, i.e., the register_sw, to store the end timestamp of each SW. When a packet arrival timestamp exceeds the value stored in the register_sw, we update the register_sw to the packet arrival timestamp plus the value of S_{SW}. Meanwhile, we set the flag, i.e., sw_completed, to 1.

B. Feature Computation

During the DW^{k}, i.e., the k-th DW, we indicate the two sequences about Aggr_{TB} and Aggr_{B} respectively as:

$$\text{DW}^{\text{k}}[0] =< \text{Aggr}_{\text{TB}}^{0}, \text{Aggr}_{\text{TB}}^{1}, \ldots, \text{Aggr}_{\text{TB}}^{\text{i}}, \ldots, \text{Aggr}_{\text{TB}}^{\text{S}_{\text{DW}}-1} > \tag{25}$$

$$\text{DW}^{\text{k}}[1] =< \text{Aggr}_{\text{B}}^{0}, \text{Aggr}_{\text{B}}^{1}, \ldots, \text{Aggr}_{\text{B}}^{\text{i}}, \ldots, \text{Aggr}_{\text{B}}^{\text{S}_{\text{DW}}-1} > \tag{26}$$

Here, $\text{Aggr}_{\text{TB}}^{\text{i}}$ and $\text{Aggr}_{\text{B}}^{\text{i}}$ are both sampled by the $\text{SW}^{\text{k,i}}$, i.e., the i-th SW in the DW^{k}.

On one hand, the time domain statistical features we extract from $\text{DW}^{\text{k}}[0]$ include information entropy and mean. On the other hand, we apply discrete wavelet transform (DWT) to $\text{DW}^{\text{k}}[1]$ and obtain the high-frequency and low-frequency

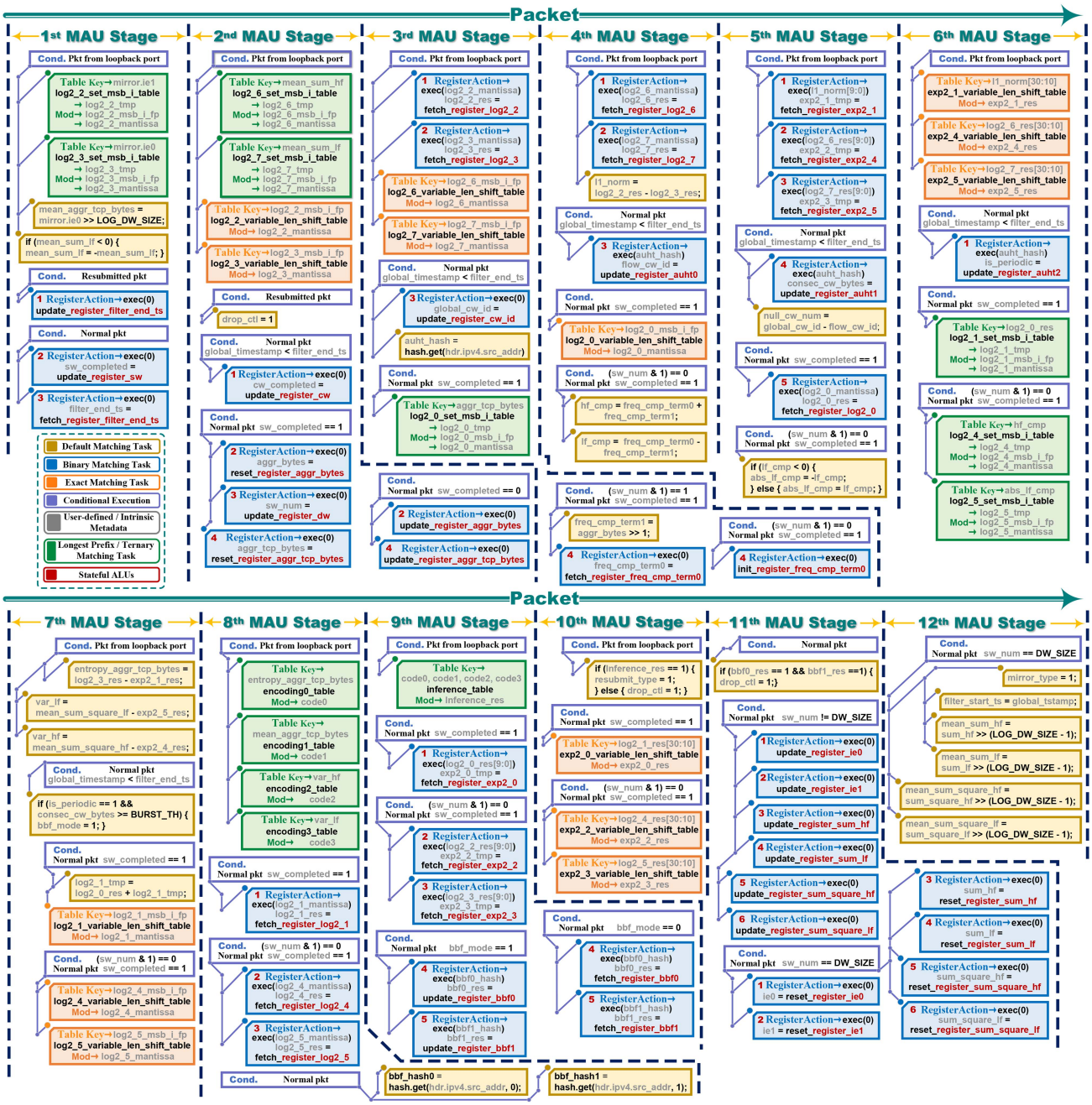


Fig. 19. The Per-packet Processing Pipeline of PLUTO.

component sequences, i.e., $\mathrm{Freq}_{\mathrm{high}}$ and $\mathrm{Freq}_{\mathrm{low}}$:

$$\mathrm{Freq}_{\mathrm{high}} =< \mathrm{Comp}^{0}_{\mathrm{high}}, \ldots, \mathrm{Comp}^{\mathrm{j}}_{\mathrm{high}}, \ldots, \mathrm{Comp}^{(\mathrm{S_{DW}}/2)-1}_{\mathrm{high}} > \tag{27}$$

$$\mathrm{Freq}_{\mathrm{low}} =< \mathrm{Comp}^{0}_{\mathrm{low}}, \ldots, \mathrm{Comp}^{\mathrm{j}}_{\mathrm{low}}, \ldots, \mathrm{Comp}^{(\mathrm{S_{DW}}/2)-1}_{\mathrm{low}} > \tag{28}$$

Here, $\mathrm{Comp}^{\mathrm{j}}_{\mathrm{high}}$ and $\mathrm{Comp}^{\mathrm{j}}_{\mathrm{high}}$ are computed by:

$$\begin{bmatrix} \mathrm{Comp}^{\mathrm{j}}_{\mathrm{high}} \\ \mathrm{Comp}^{\mathrm{j}}_{\mathrm{low}} \end{bmatrix} = \begin{bmatrix} \mathrm{f_h}[0] & \mathrm{f_h}[1] \\ \mathrm{f_l}[0] & \mathrm{f_l}[1] \end{bmatrix} \cdot \begin{bmatrix} \mathrm{Aggr}^{2\mathrm{j}}_{\mathrm{B}} \\ \mathrm{Aggr}^{2\mathrm{j}+1}_{\mathrm{B}} \end{bmatrix} \tag{29}$$

Note that, $\mathrm{f_h}$ and $\mathrm{f_l}$ are both 2D vectors, acting as high-pass and low-pass filters respectively. Furthermore, we separately extract the variances from $\mathrm{Freq}_{\mathrm{high}}$ and $\mathrm{Freq}_{\mathrm{low}}$.

1) Discrete Wavelet Transform: To facilitate conducting the DWT in the P4 switch, we set $\mathrm{S_{DW}}$ to the power of 2. Meanwhile, we set $\mathrm{f_0}$ and $\mathrm{f_1}$ to $< 0.5, 0.5 >$ and $< 0.5, -0.5 >$ respectively. Therefore, we can use fixed-length shift to compute (29). As shown in Fig. 21, we present the P4 pseudocode for the DWT.

When computing the j-th pair of frequency components, we need to utilize both $\mathrm{Aggr}^{2\mathrm{j}}_{\mathrm{B}}$ and $\mathrm{Aggr}^{2\mathrm{j}+1}_{\mathrm{B}}$. When $\mathrm{SW}^{\mathrm{k},2\mathrm{j}}$ is completed, we use the register_freq_cmp_term0 to store the value of $(\mathrm{Aggr}^{2\mathrm{j}}_{\mathrm{B}} \gg 1)$. When $\mathrm{SW}^{\mathrm{k},2\mathrm{j}+1}$ is completed, we fetch the value of register_freq_cmp_term0 and compute the value of

```
#define STAT(name,width,obj) Register ⟨bit ⟨width⟩, bit ⟨16⟩⟩ (1) register_##name; \
RegisterAction ⟨bit ⟨width⟩, bit ⟨16⟩, bit ⟨width⟩⟩ (register_##name) update_register_##name = { \
    void apply (inout bit ⟨width⟩ value, out bit ⟨width⟩ read_value) { \
        value = value + (bit ⟨width⟩) ig_md.##obj; } } \
RegisterAction ⟨bit ⟨width⟩, bit ⟨16⟩, bit ⟨width⟩⟩ (register_##name) reset_register_##name = { \
    void apply (inout bit ⟨width⟩ value, out bit ⟨width⟩ read_value) { \
        read_value = value + (bit ⟨width⟩) ig_md.##obj; value = 0; } }
STAT(aggr_bytes, 32, ipv4_tcp_len) STAT(aggr_tcp_bytes, 32, ipv4_len)
Register ⟨bit ⟨48⟩, bit ⟨16⟩⟩ (1) register_sw;
RegisterAction ⟨bit ⟨48⟩, bit ⟨16⟩, bit ⟨1⟩⟩ (register_sw) update_register_sw = {
    void apply (inout bit ⟨48⟩ value, out bit ⟨1⟩ read_value) {
        if (ig_prsr_md.global_tstamp > value) {
            value = ig_prsr_md.global_tstamp + SW_SIZE; read_value = 1;
        } else { read_value = 0; } } }
Register ⟨bit ⟨16⟩, bit ⟨16⟩⟩ (1) register_dw;
RegisterAction ⟨bit ⟨16⟩, bit ⟨16⟩, bit ⟨1⟩⟩ (register_dw) update_register_dw = {
    void apply (inout bit ⟨16⟩ value, out bit ⟨1⟩ read_value) {
        if (DW_SIZE - 1 != value) { value = value + 1; read_value = 0;
        } else { value = 0; read_value = 1; } } }
apply { // Maintain the Sampling Window (SW)
    ig_md.sw_completed = update_register_sw.execute(0); // 1st Stage
    if (ig_md.sw_completed == 1) { // An SW is completed, Update the number of SWs in current DW
        ig_md.sw_num = update_register_dw.execute(0); // 2nd Stage
        ig_md.sw_aggr_bytes = reset_register_aggr_bytes(0); // 2nd Stage
        ig_md.sw_aggr_tcp_bytes = reset_register_aggr_tcp_bytes(0); // 2nd Stage
    } else { update_register_aggr_bytes(0); update_register_aggr_tcp_bytes(0); /* 2nd Stage */ } }
```

Fig. 20. P4 Pseudocode for Time Window.

```
Register ⟨bit ⟨32⟩, bit ⟨16⟩⟩ (1) register_freq_cmp_term0;
RegisterAction ⟨bit ⟨32⟩, bit ⟨16⟩, bit ⟨32⟩⟩ (register_freq_cmp_term0) init_register_freq_cmp_term0
= { void apply (inout bit ⟨32⟩ value, out bit ⟨32⟩ read_value) { value = ig_md.sw_aggr_bytes ≫ 1; } }
RegisterAction ⟨bit ⟨32⟩, bit ⟨16⟩, bit ⟨32⟩⟩ (register_freq_cmp_term0) fetch_register_freq_cmp_term0
= { void apply (inout bit ⟨32⟩ value, out bit ⟨32⟩ read_value) { read_value = value; } }
apply {
    ..... // Compute frequency components of DWT when an SW is completed.
    if ((ig_md.sw_num & 8w1) == 8w0) { // the number of SW is an even.
        init_register_freq_cmp_term0.execute(0); // 3rd Stage
    } else { // The number of SW is an odd.
        ig_md.freq_cmp_term0 = fetch_register_freq_cmp_term0(0); // 3rd Stage
        ig_md.freq_cmp_term1 = ig_md.sw_aggr_bytes ≫ 1; // 3rd Stage
        ig_md.hf_cmp = ig_md.freq_cmp_term0 + ig_md.freq_cmp_term1; // 4th Stage
        ig_md.lf_cmp = ig_md.freq_cmp_term0 − ig_md.freq_cmp_term1; /* 4th Stage */ } }
```

Fig. 21. P4 Pseudocode for Conducting the DWT.

$(\mathrm{Aggr}_{\mathrm{B}}^{2j+1} \gg 1)$. Furthermore, by the sum and difference of the two values, we obtain the $\mathrm{Comp}_{\mathrm{high}}^{\mathrm{j}}$ and $\mathrm{Comp}_{\mathrm{low}}^{\mathrm{j}}$ respectively.

2) Statistical Feature Computation: Since we can compute the mean through fixed-length shift, we will focus on introducing the computation of both information entropy and variance in the P4 switch by utilizing our P4 Function Tool.

Information Entropy: We compute the information entropy of the $\mathrm{DW}^{\mathrm{k}}[0]$ and indicate it as $\mathrm{IE}(\mathrm{Aggr}_{\mathrm{TB}})$:

$$\mathrm{IE}(\mathrm{Aggr}_{\mathrm{TB}}) = \mathrm{IE}^0 - (\mathrm{IE}^1 \,/\, \mathrm{IE}^0) \tag{30}$$

$$\mathrm{IE}^0 = \sum_{\mathrm{i}=0}^{\mathrm{S_{DW}}} \mathrm{Aggr}_{\mathrm{TB}}^{\mathrm{i}}, \mathrm{IE}^1 = \sum_{\mathrm{i}=0}^{\mathrm{S_{DW}}} [\log_2(\mathrm{Aggr}_{\mathrm{TB}}^{\mathrm{i}}) \cdot \mathrm{Aggr}_{\mathrm{TB}}^{\mathrm{i}}] \tag{31}$$

We utilize the register_ie0 and register_ie1 to store the stateful variables, i.e., IE^0 and IE^1 respectively. When the $\mathrm{SW}^{\mathrm{k,i}}$ is completed, we update both register_ie0 and register_ie1. In particular, with our P4 Function Tool, we utilize the function operations including the Logarithm and Exponent to compute the value of $[\log_2(\mathrm{Aggr}_{\mathrm{TB}}^{\mathrm{i}}) \cdot \mathrm{Aggr}_{\mathrm{TB}}^{\mathrm{i}}]$:

$$\log_2(\mathrm{Aggr}_{\mathrm{TB}}^{\mathrm{i}}) \cdot \mathrm{Aggr}_{\mathrm{TB}}^{\mathrm{i}} = 2^{[\log_2(\mathrm{Aggr}_{\mathrm{TB}}^{\mathrm{i}}) + \log_2(\log_2(\mathrm{Aggr}_{\mathrm{TB}}^{\mathrm{i}}))]} \tag{32}$$

When the DW^{k} is completed, we compute the value of $(\mathrm{IE}^1 \,/\, \mathrm{IE}^0)$ as $2^{[\log_2(\mathrm{IE}^1) - \log_2(\mathrm{IE}^0)]}$. As shown in Fig. 22, we demonstrate the P4 pseudocode for computing the information entropy. Notably, constrained by the 12 MAU stages, when the DW^{k} is completed, we generate a mirrored packet and emit it to the loopback port. This packet will re-enter the ingress pipeline to finish computing (30) and execute the pre-inference.

Variance: We compute the variances of $\mathrm{Freq}_{\mathrm{high}}$ and $\mathrm{Freq}_{\mathrm{low}}$ respectively. Take the variance of $\mathrm{Freq}_{\mathrm{high}}$ as an example, to

```
STAT(ie0, 32, sw_aggr_tcp_bytes) STAT(ie1, 48, exp_res) #define FRAC_WIDTH 10
apply { /********** If current packet is a normal packet, execute below P4 codes **********/
    log2_0_set_msb_i_table.apply(); /* 3rd Stage */ log2_0_variable_len_shift_table.apply(); /* 4th Stage */
    ig_md.log2_0_res = fetch_register_log2_0.execute(ig_md.log2_0_mantissa); // 5th Stage
    log2_1_set_msb_i_table.apply(); /* 6th Stage */ log2_1_variable_len_shift_table.apply(); /* 7th Stage */
    ig_md.log2_1_tmp = ig_md.log2_0_res + ig_md.log2_1_tmp; // 7th Stage
    ig_md.log2_1_res = fetch_register_log2_1.execute(ig_md.log2_1_mantissa); // 8th Stage
    ig_md.exp2_0_res = fetch_register_exp2_0.execute((bit ⟨32⟩)ig_md.log2_1_res[9 : 0]); // 9th Stage
    exp2_0_variable_len_shift_table.apply(); // 10th Stage
    if (ig_md.dw_completed == 0) { update_register_ie0.execute(0); update_register_ie1.execute(0);
    } else { ig_md.ie0 = reset_register_ie0.execute(0); ig_md.ie1 = reset_register_ie1.execute(0);
        ig_dprsr_md.mirror_type = MIRROR_TYPE_I2E; } // 11~13 lines are all executed in 11th Stage
    /******* If current packet is a mirrored packet from loopback port, execute below P4 codes *******/
    log2_2_set_msb_i_table.apply(); log2_3_set_msb_i_table.apply(); // 1st Stage
    log2_2_variable_len_shift_table.apply(); log2_3_variable_len_shift_table.apply(); // 2nd Stage
    ig_md.log2_2_res = fetch_register_log2_2.execute(ig_md.log2_2_mantissa); // 3rd Stage
    ig_md.log2_3_res = fetch_register_log2_3.execute(ig_md.log2_3_mantissa); // 3rd Stage
    ig_md.l1_norm = ig_md.log2_2_res − ig_md.log2_3_res; // 4th Stage
    ig_md.exp2_1_res = fetch_register_exp2_1.execute((bit ⟨32⟩)ig_md.l1_norm[9 : 0]); // 5th Stage
    exp2_1_variable_len_shift_table.apply(); // 6th Stage
    ig_md.entropy_aggr_tcp_bytes = ig_md.log2_3_res − ig_md.exp2_1_res; /* 7th Stage */ }
```

Fig. 22. P4 Pseudocode for Conducting the Information Entropy.

```
STAT(sum_square_hf, 48, exp2_2_res) STAT(sum_hf, 48, hf_cmp) #define FRAC_WIDTH 10
apply { /********** If current packet is a normal packet, execute below P4 codes **********/
    log2_4_set_msb_i_table.apply(); /* 6th Stage */ log2_4_variable_len_shift_table.apply(); /* 7th Stage */
    ig_md.log2_4_res = fetch_register_log2_4.execute((bit ⟨32⟩) ig_md.log2_4_mantissa); // 8th Stage
    ig_md.exp2_2_res = fetch_register_exp2_2.execute((bit ⟨32⟩) ig_md.log2_4_res[9:0]); // 9th Stage
    exp2_2_variable_len_shift_table.apply(); // 10th Stage
    if (ig_md.dw_completed == 0) {
        update_register_sum_square_hf.execute(0); update_register_sum_hf.execute(0);
    } else { ig_md.sum_square_hf = reset_register_sum_square_hf.execute(0);
        ig_md.sum_hf = reset_register_sum_hf.execute(0);
        ig_dprsr_md.mirror_type = MIRROR_TYPE_I2E; } // 8~12 lines are all executed in 11th Stage
    /******* If current packet is a mirrored packet from loopback port, execute below P4 codes *******/
    log2_6_set_msb_i_table.apply(); /* 2nd Stage */ log2_6_variable_len_shift_table.apply(); /* 3rd Stage */
    fetch_register_log2_6.execute((bit ⟨32⟩)ig_md.log2_6_mantissa); // 4th Stage
    fetch_register_exp2_4.execute((bit ⟨32⟩)ig_md.log2_6_res[9:0]); // 5th Stage
    exp2_4_variable_len_shift_table.apply(); // 6th Stage
    ig_md.var_hf = ig_md.mean_sum_square_hf − ig_md.exp2_4_res; /* 7th Stage */ }
```

Fig. 23. P4 Pseudocode for Conducting the Variance.

compute it, we have:

$$\mathrm{Var}(\mathrm{Freq}_{\mathrm{high}}) = \mathrm{V}^0 - (\mathrm{V}^1)^2, \mathrm{N_C} = (\mathrm{S_{DW}}/2) - 1 \tag{33}$$

$$\mathrm{V}^0 = \frac{2}{\mathrm{S_{DW}}} \sum_{\mathrm{j}=0}^{\mathrm{N_C}} (\mathrm{Comp}_{\mathrm{high}}^{\mathrm{j}})^2,$$

$$V^1 = \frac{2}{\mathrm{S_{DW}}} \sum_{\mathrm{j}=0}^{\mathrm{N_C}} \mathrm{Comp}_{\mathrm{high}}^{\mathrm{j}} \tag{34}$$

We use the register_sum_square_hf and register_sum_hf saparately to store stateful variables, i.e., V^0 and V^1. Besides, we update the two register when an SW is completed. By the P4 Function Tool, we use the Logarithm and Exponent to compute the values of both $(\mathrm{Comp}_{\mathrm{high}}^{\mathrm{j}})^2$ and $(\mathrm{V}^1)^2$:

$$(\mathrm{Comp}_{\mathrm{high}}^{\mathrm{j}})^2 = 2^{2 \cdot \log_2(\mathrm{Comp}_{\mathrm{high}}^{\mathrm{j}})}, (\mathrm{V}^1)^2 = 2^{2 \cdot \log_2(\mathrm{V}^1)} \tag{35}$$

As shown in Fig. 23, we present the P4 pseudocode of computing the variance. Notably, for the BM task of Logarithm, we amplify the value stored in the register entry by a factor of two offline. Besides, the computation of (33) is finished by the mirrored packet mentioned before.

C. Pre-Inference Model Mapping

We utilize four LPM tasks to separately encode four extracted features, including the information entropy of $\mathrm{DW}^{\mathrm{k}}[0]$, the mean of $\mathrm{DW}^{\mathrm{k}}[0]$, the variance of $\mathrm{Freq}_{\mathrm{high}}$, and the variance of $\mathrm{Freq}_{\mathrm{low}}$. Besides, for executing the inference, we leverage a TM task containing four keys to conduct the joint matching. Each key corresponds to a feature code. The relevant P4 pseudocode is demonstrated in Fig. 24.

D. Flow-Based Attacker Filtering and Blocklist

When the pre-inference result is 1, we will enable the TPSM for 1 minute to apply the AUHT. We implement the Counting

TABLE VI
RELATIVE ERROR STATISITCS OF P4 FUNCTION TOOL AND BASELINE

Methods	P4-Function Tool				Flex Switch Libs (m=6)				Flex Switch Libs (m=8)				Flex Switch Libs (m=10)			
Stats (%)	μ	σ	$\mu+3\sigma$	CI Setup	μ	σ	$\mu+3\sigma$	CI Setup	μ	σ	$\mu+3\sigma$	CI Setup	μ	σ	$\mu+3\sigma$	CI Setup
32 bits (FRAC_WIDTH = 10)																
Log_2	5.01e-5	2.24e-5	1.17e-4	[0,1.17e-4]	4.04e-4	2.60e-4	1.18e-3	[0,1.18e-3]	1.03e-4	6.76e-5	3.06e-4	[0,3.06e-4]	3.40e-5	2.44e-5	1.07e-4	[0,1.07e-4]
Sqrt	1.70e-4	1.05e-4	4.86e-4	[0,4.86e-4]	2.71e-3	1.69e-3	7.77e-3	[0,7.77e-3]	6.77e-4	4.21e-4	1.94e-3	[0,1.94e-3]	1.69e-4	1.05e-4	4.85e-4	[0,4.85e-4]
Sin	0.18	13.82	41.63	[0,15.00]	6.04	521.80	1571.43	[0,15.00]	5.72	345.85	1043.26	[0,15.00]	5.81	403.24	1215.53	[0,15.00]
Cos	0.22	38.40	115.42	[0,15.00]	5.99	401.05	1209.13	[0,15.00]	6.66	851.05	2559.82	[0,15.00]	6.96	1172.58	3524.69	[0,15.00]
64 bits (FRAC_WIDTH = 12)																
Log_2	5.84e-6	3.81e-6	1.73e-5	[0,1.73e-5]	1.91e-4	1.65e-4	6.85e-4	[0,6.85e-4]	4.77e-5	4.14e-5	1.72e-4	[0,1.72e-4]	1.22e-5	1.05e-5	4.36e-5	[0,4.36e-5]
Sqrt	4.18e-5	2.55e-5	1.18e-4	[0,1.18e-4]	2.73e-3	1.72e-3	7.87e-3	[0,7.87e-3]	6.82e-4	4.29e-4	1.97e-3	[0,1.97e-3]	1.71e-4	1.07e-4	4.92e-4	[0,4.92e-4]

[1] We use blue to mark the **best** results and use orange to mark the **worst** result.

```
#define ENCODING(feature,i) \
action set_code##i(bit<8> p0) { ig_md.code##i = p0; } \
table encoding####_table { key={ ig_md.feature:lpm; } actions={ set_code##i; } }
action set_pre_inference_res(bit<1> p0) {ig_md.pre_inference_res = p0; }
table inference_table {
    key={ ig_md.code0:ternary; ig_md.code1:ternary; ig_md.code2:ternary; ig_md.code3:ternary; }
    actions={ set_inference_res; } }
ENCODING(entropy_aggr_tcp_bytes,0) ENCODING(mean_aggr_tcp_bytes,1)
ENCODING(var_hf,2) ENCODING(var_lf,3)
apply {
    encoding0_table.apply(); encoding1_table.apply(); encoding2_table.apply(); // 8th Stage
    encoding3_table.apply(); /* 8th Stage */ inference_table.apply(); /* 9th Stage */ }
```

Fig. 24. P4 Pseudocode for Pre-inference Model Mapping.

Window(CW) as the same with the SW and adopt the CRC32 as the hashing algorithm. Each hash array (32-bit register) in the AUHT contains 2^{17} entries. Therefore, the AUHT occupies 1.573 MB SRAM. Besides, we use $\text{register}_{\text{f}}\text{ilter}_{\text{e}}\text{nd}_{\text{t}}\text{s}$ as the timer, it stores the end timestamp of the TPSM. When the packet arrival timestamp exceeds the end timestamp, the TPSM will shut down.

In addition, we implement the blocklist based on the blocked bloom filter. We split the one-hash-array bloom filter into two 1-bit registers. Each register contains 2^{20} entries, thus the blocked bloom filter occupies 0.262 MB SRAM.

E. Local CPU Control Logic

At the very beginning of the runtime PLUTO prototype, the local CPU of the P4 switch is responsible for 1) pre-installing table entries and pre-write register entries for computing the Logarithm and Exponent, 2) pre-installing table entries for executing the pre-inference model.

VII. EVALUATION

In this section, we evaluate the performance of PLUTO. The experimental results will answer the issues below:

1) Does the P4 Function Tool have lower errors and TCAM usage compared with the baseline? (Section VII-A)
2) Does PLUTO have stronger detection performance and lower mitigation response time compared with the baseline? (Section VII-B)

A. P4 Function Tool Evaluation

Baseline: We use the state-of-the-art solution for achieving function operations in P4, namely Flex Switch Libs [37], as the baseline. This solution uses the longest prefix encoding (LPE) to achieve function operations in the P4 switch. Here, the width of LPE is indicated as m. We compare the P4 Function Tool with the Flex Switch Libs under the configurations of $\text{m} = 6$, $\text{m} = 8$, and $\text{m} = 10$ respectively.

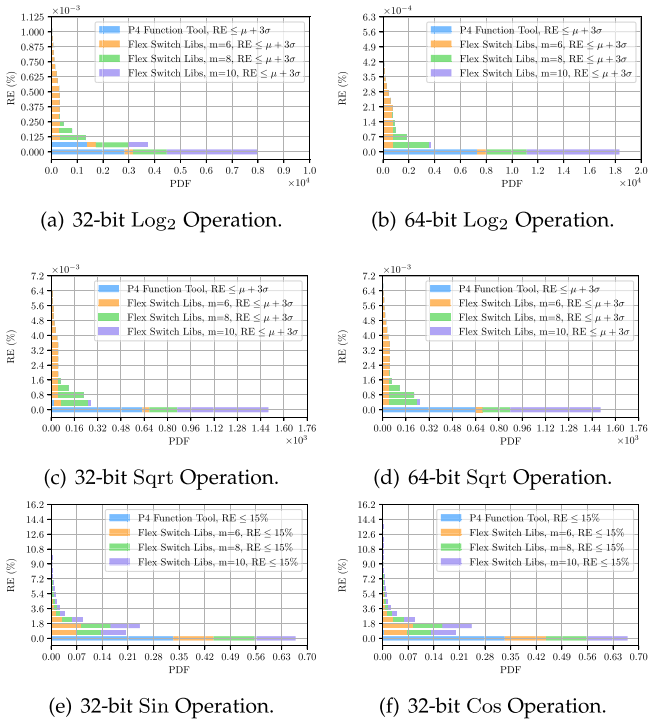
Experimental Setup: To facilitate the accuracy evaluation for the P4 Function Tool, we use C++ to implement the match/action mode of the P4 switch. Based on this, we employ the C++ program to evaluate the accuracy of P4 Function Tool by using extensive 32-bit and 64-bit input data respectively. For 32-bit input data, we set FRAC_WIDTH to 10. For 64-bit input data, we set FRAC_WIDTH to 12. We enter the data in the stride of $(2 + 2^{-\text{FRAC\_WIDTH}})$, starting from the minimal value to the maximum value demonstrated in Table V. Notably, for 64-bit input data, since its scope is too wide, the stride is expanded to $2 \cdot \text{stride} + 1$ after every $2^{\text{FRAC\_WIDTH}}$ input rounds.

Accuracy Comparison: As shown in Table VI, we measure the relative errors (REs) caused by the P4 Function Tool to evaluate its accuracy. We use extensive input data (32-bit input data and 64-bit input data) for the RE measurement. We compute the RE statistics including mean (μ), standard deviation (σ). The results of RE statistics indicate that the accuracy of our P4 Function Tool is higher than the Flex Switch Libs overall. Additionally, we measure the RE distribution of the P4 Function Tool. According to the three-sigma principle, we set the confidence interval (CI) as $[0, \mu + 3\sigma]$. Notably, for both Sin and Cos, the relevant RE distribution of the baseline significantly deviates from a normal Gaussian distribution, thus we uniformly set the CI as $[0, 15]$ for the two functions.

As shown in Fig. 25, we demonstrate the RE distribution within the CI. For four different functions (Log_2, Sqrt, Sin, and Cos), the REs caused by the P4 Function Tool are overall more stable and closer to zero compared with the baseline.

Notably, the baseline utilizes the LPE to implement the function operations, except for the Exponent (2^{x}). For 2^{x}, the baseline uses an exact matching task to achieve it, resulting in the same accuracy as the P4 Function Tool when computing 2^{x}. Therefore, for 2^{x}, we compare only the memory usage between the P4 Function Tool and the baseline. Additionally, although the accuracy of Flex Switch Libs (m=10) is close to the Function Tool in several cases, the TCAM usage of Flex Switch Libs (m=10) is significantly higher than the P4 Function Tool, which we will demonstrate in the following content.

Memory Usage Comparison: As shown in Table VII, we present the memory (SRAM and TCAM) usage of the P4 Function Tool. Compared with the baseline, the P4 Function Tool significantly reduces the usage of expensive TCAM by an average of 90.51% for 32-bit input data and 98.53% for 64-bit input data. In addition, the P4 Function Tool occupies less than 0.053% of the total SRAM (120 MB for Intel Tofino).



(a) 32-bit Log_2 Operation. (b) 64-bit Log_2 Operation.

(c) 32-bit Sqrt Operation. (d) 64-bit Sqrt Operation.

(e) 32-bit Sin Operation. (f) 32-bit Cos Operation.

Fig. 25. RE Distribution within Confidence Interval.

TABLE VII
MEMORY USAGE OF P4 FUNCTION TOOL AND BASELINE

Methods	P4-Function Tool					Flex Switch Libs (m=6)		Flex Switch Libs (m=8)		Flex Switch Libs (m=10)	
32-bit (FRAC_WIDTH = 10)											
Function Operation	Log_2	Sqrt	Sin	Cos	Exp_2	Log_2/Sqrt Sin/Cos	Exp_2	Log_2/Sqrt Sin/Cos	Exp_2	Log_2/Sqrt Sin/Cos	Exp_2
TCAM (KB) ▼90.51%	0.12 ▼96.41%		1.16 ▼84.60%		-	3.37	-	12.50	-	46.00	-
SRAM (KB) < 0.012%	4.36 < 0.01%	4.49 < 0.01%	13.76 < 0.012%		8.25 ▼93.35% < 0.01%		123.99		123.99		123.99
64-bit (FRAC_WIDTH = 12)											
Function Operation	Log_2		Sqrt		Exp_2	Log_2/Sqrt	Exp_2	Log_2/Sqrt	Exp_2	Log_2/Sqrt	Exp_2
TCAM (KB) ▼98.53%	0.49 ▼98.53%				-	14.74	-	56.99	-	219.99	-
SRAM (KB) < 0.053%	33.48 < 0.03%		33.98 < 0.03%		65.00 ▼95.78% < 0.053%		2015.98		2015.98		2015.98

[1] ▼ represent the decrease of P4 Function Tool relative to the baseline.
[2] We use red to mark the proportion of P4 Function Tool's SRAM usage to the total SRAM of Intel Tofino (120MB).

B. Detection and Mitigation Evaluation

Baselines: To evaluate the performance improvement of detection and mitigation brought by PLUTO, we establish three baselines:

- *P&F:* It is an SDN-based solution for defending LDoS attacks. It adopts the time window as units to analyze whether a flow table entry corresponds to an LDoS attack flow. We prototype P&F by the Ryu 4.34 [45] based on its relevant paper [6].
- *Whisper:* It is an Intel DPDK-based system [7] for detecting malicious traffic. We build its open source project and only tune its hyper-parameters for acceptable performance.
- *NetBeacon:* It is a P4-based per-flow ML inference solution [20]. We configure its task to distinguish LDoS attack flows from other flows. To deploy it on the BMv2 switch, we convert its open source $\mathrm{P4}_{16}$ codes from the TNA style to the V1Model style.

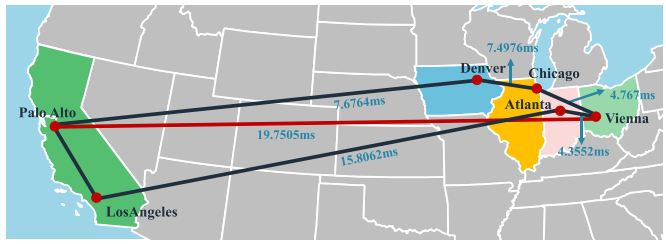


Fig. 26. The Network Topology of Epoch.

Testbed: We build a testbed consisting of Ubuntu 20.04 LTS operating system (Linux 5.4.0), one Intel Xeon E5-2680 v4 CPU (32 GB RAM and 16 cores), as well as one Intel I210 NIC (1 Gbps, one port with 4 RX queues, and supporting Intel DPDK).

With our testbed, we utilize the network container Mininet 2.3.0 [46] to run a real-world network topology, evaluating the detection and mitigation performance of PLUTO, P&F, and NetBeacon. Since Whisper is a virtualization module in a software-defined middlebox, it cannot interfere with traffic for mitigation. Therefore, we only evaluate its detection performance by an end-to-end approach. We utilize the testbed for deploying Whisper and use another server for generating traffic.

Real-world Topology: As shown in Fig. 26, we utilize a real-world network topology named Epoch for evaluating [47]. It is included in the Internet Topology Zoo dataset [48].

In this topology, each city is represented by a switch. The links between a local user and a city switch, i.e., the local links, have a bandwidth of 1 Gbps and a delay of 0 ms. In addition, the links connecting any two cities, i.e., the city links, have a bandwidth of 1 Gbps. And their respective link delays are marked in Fig. 26. In the following content, we use the city name to represent its revalant switch.

Within our evaluation, we assume there are attackers in the Palo Alto city, they launch LDoS attacks targeting the local users in the Vienna city. As a result, the available TCP bandwidth in the red link is deteriorated.

To defend against LDoS attacks, we deploy the prototype of PLUTO and the baselines (P&F and NetBeacon) for the Palo Alto. Concretely, Under the cases of PLUTO and NetBeacon, Palo Alto is a BMv2 switch, and we load the compiled file of P4 code to it. Under the case of P&F, Palo Alto is an OpenvSwitch connected to a Ryu controller where the program of P&F is running.

Real-world Traffic Dataset: We replay the real-world traffic datasets, which are collected from the WIDE MAWI Gigabit backbone network [44], as the background traffic in the city link between Palo Alto and Vienna.

Within our evaluation, we establish multiple traffic scenarios with different TCP packet proportions. As shown in Table VIII, we analyze the TCP packet proportion in the real-world traffic datasets collected from 2022.7.1 to 2022.7.31. The TCP packet proportion fluctuates from 0.55 to 0.85.

By Increasing the TCP packet proportion according to the step of 0.1, we select the traffic datasets of the 23 rd day, the 8th day, the 3 rd day, the 18th day, and the 17th day to establish

TABLE VIII
THE PROPORTION OF TCP PACKETS IN REAL-WORLD TRAFFIC DATASETS [44]

1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
0.643	0.666	0.700	0.622	0.620	0.645	0.628	0.611	0.683	0.747
11th	**12th**	**13th**	**14th**	**15th**	**16th**	**17th**	**18th**	**19th**	**20th**
0.639	0.668	0.644	0.692	0.647	0.772	0.851	0.793	0.708	0.720
21st	**22nd**	**23rd**	**24th**	**25th**	**26th**	**27th**	**28th**	**29th**	**30th**
0.700	0.727	0.553	0.670	0.561	0.644	0.749	0.762	0.754	0.819

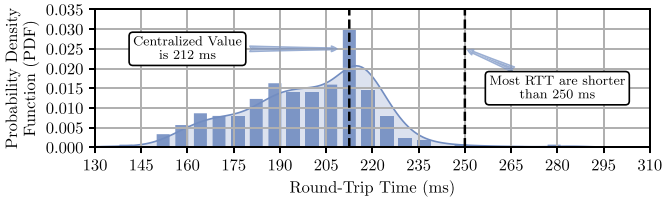


Fig. 27. The Measurement of Round-trip Time in Epoch Network.

TABLE IX
CONFIGURATION OF LDOS ATTACK PARAMETERS

Detection Testing				Mitigation Testing			
Group Number	Pulse Duration (ms)	Group Number	Pulse Duration (ms)	Group Number	Pulse Duration (ms)	Group Number	Pulse Duration (ms)
0	212	1	216	0	214	1	218
2	220	3	224	2	222	3	226
4	228	5	232	4	230	5	234
6	236	7	240	6	238	7	242
8	244	9	248	8	246	9	250

[1]The Attack Period and Attack Intensity of all groups are set to 1s and 45Mbps respectively.

five traffic scenarios. Their TCP packet proportions are 0.553, 0.611, 0.7, 0.793, and 0.851 respectively.

LDoS Attack Setup: LDoS attacks contain three parameters: attack period, attack intensity, and burst duration. Based on the recommendation of [12], the attack period should be set as minRTO which is default to 1s according to RFC 2988 [13]; the attack intensity should reach the bottleneck link bandwidth (45Mbps); and the burst duration should cover the RTT of the bottleneck link.

To set the burst duration, as shown in Fig. 27, we measure the RTT of the link connecting Palo Alto to Vienna (the red link in Fig. 26), 500 times. In this context, the average RTT is 212 ms, and most RTTs are short than 250 ms. Therefore, for the detection testing, we set the burst duration in steps of 4 ms between the interval, i.e., [212 ms, 248ms]; for the mitigation testing, we set the burst duration in steps of 4 ms between the interval, i.e., [214 ms, 250ms]. All groups of the LDoS attack parameter are shown in Table IX.

In addition, we utilize an IP pool of size 72, and the source IP of an LDoS attack packet is randomly generated from the IP pool. Therefore, the byte rate of each LDoS attack flow is 0.625 Mbps.

Feature Record Collection of PLUTO: In each traffic scenario, we set up a normal scenario (without LDoS attacks) and an attack scenario (with LDoS attacks) to separately obtain benign and malicious feature records for training the pre-inference model. Notably, each feature record contains the statistical features extracted from a DW.

In normal scenarios, we only replay the real-world traffic dataset as background traffic. In particular, since the real-world traffic dataset is recorded offline, its TCP traffic is stateless. Therefore, in the attack scenario, to ensure LDoS attacks are effective, we only replay the UDP traffic in the real-world traffic dataset. Meanwhile, we utilize the Iperf to send multiple TCP flows with connection states, filling the original TCP bandwidth. Additionally, we utilize the ten groups of LDoS attack parameters, listed in Table IX (focusing on the column of Detection Testing), to launch LDoS attacks in the attack scenario.

TABLE X
NUMBER OF BENIGN AND MALICIOUS FEATURE RECORDS

Case of $(S_{SW}(ms), S_{DW})$	Benign	Malicious	Ratio	Case of $(S_{SW}(ms), S_{DW})$	Benign	Malicious	Ratio
$(1, 2^3)$	1063270	1221285	47:53	$(2, 2^3)$	569130	726615	44:56
$(1, 2^4)$	535430	679965	44:56	$(2, 2^4)$	285245	374600	43:57
$(1, 2^5)$	268505	351690	43:57	$(2, 2^5)$	142865	188220	43:57
$(1, 2^6)$	133815	175890	43:57	$(2, 2^6)$	71260	94305	44:57
$(1.5, 2^3)$	749840	899360	45:55	$(2.5, 2^3)$	467695	600700	43:57
$(1.5, 2^4)$	371980	486470	43:57	$(2.5, 2^3)$	234115	306790	43:57
$(1.5, 2^5)$	187095	245525	43:57	$(2.5, 2^3)$	116775	153095	43:57
$(1.5, 2^6)$	93185	122135	43:57	$(2.5, 2^3)$	58450	76800	43:57

We collect feature records under different combinations of S_{SW} and S_{DW}. Here, $S_{SW} \in \{1ms, 1.5ms, 2ms, 2.5ms\}$ and $S_{DW} \in \{2^3, 2^4, 2^5, 2^6\}$. We indicate each combination as the format of (S_{SW}, S_{DW}).

With each case of (S_{SW}, S_{DW}), we collect feature records under five traffic scenarios respectively. For each traffic scenario, we take 400s to collect benign feature records in the normal scenario. Additionally, we take 60s to collect malicious feature records under each group of LDoS attacks. After a group of LDoS attacks stops, we spend 20s collecting benign feature records. Overall, we spend 800s and 600s separately collecting malicious and benign feature records. As shown in Table X, we present the number of benign and malicious feature records in each case of (S_{SW}, S_{DW}).

Detection Performance of PLUTO: Refering to Section V-B, we adopt three different tree-based EL algorithm to train pre-inference model. We compare the detection performance between them by four metrics: 1) Recall, 2) the area under ROC curve (AUC), 3) F1 Score, and 4) equal error rate (EER).

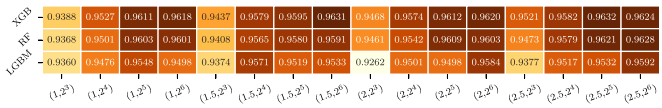
With each case of (S_{SW}, S_{DW}), we utilize the relavant collected feature records to train and test the pre-inference model. We split all feature records into training records and testing records based on the ratio of 1:1. As shown in Fig. 28, we present the measured metrics of the pre-inference model under different cases of (S_{SW}, S_{DW}).

When S_{DW} increases to 2^5, the detection performance of PLUTO reaches an acceptable level. As S_{DW} increases to 2^6, the detection performance does not improve significantly, or even decreases slightly.

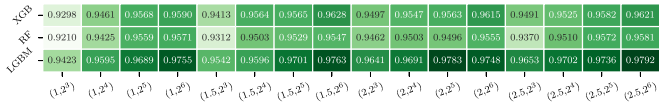
In addition, for the three tree-based EL algorithms, XGB has the optimal metrics of AUC (0.9579), F1 Score (0.9632), and EER (0.0421). Its higher F1 Score indicates that it has a strong ability to classify both benign and malicious feature records. While LGBM has the optimal Recall (0.9792). Since its Recall is significantly higher than its F1 Score, LGBM has a stronger classification ability for malicious feature records than for benign feature records.

Overall, with appropriate configuration, PLUTO can achieve AUC (0.9579), F1 Score (0.9632), EER (0.0421), and Recall (0.9792), the best.

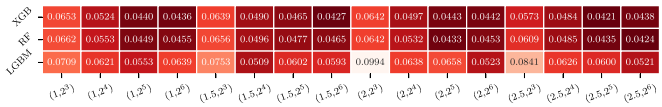
Detection Performance Comparison: We measure the detection performance of baselines (P&F, Whisper, and NetBeacon) under the same five traffic scenarios. Notably, we use the same



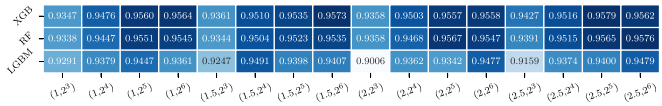
(a) The F1 Score Metric under Different Cases of (S_{SW}(ms), S_{DW}).



(b) The Recall Metric under Different Cases of (S_{SW}(ms), S_{DW}).



(c) The EER Metric under Different Cases of (S_{SW}(ms), S_{DW}).



(d) The AUC Metric under Different Cases of (S_{SW}(ms), S_{DW}).

Fig. 28. Detection Performance of PLUTO under Different Tree-based EL Algorithms.

TABLE XI
DETECTION PERFORMANCE COMPARISON

Method	AUC	F1 Score	EER	Recall
PLUTO	0.9579 ▲1.83%	0.9632 ▲7.27%	0.0421 ▼27.96%	0.9792 ▲9.58%
Whisper	0.9500 ▼0.82%	0.9026 ▼6.29%	0.0500 ▲18.76%	0.9172 ▼6.33%
P&F	0.9375 ▼2.13%	0.8932 ▼7.27%	0.0625 ▲48.46%	0.8846 ▼9.66%
NetBeacon	0.9348 ▼2.41%	0.8980 ▼6.77%	0.0652 ▲54.87%	0.8800 ▼10.13%

[1] ▲ and ▼ refer to the increase and decrease of a baseline relative to PLUTO respectively.
[2] ▲ and ▼ refer to the average increase and decrease of PLUTO relative to baselines respectively.

ten groups of LDoS attack parameter, listed in Table IX (focusing on the column of Detection Testing), to launch LDoS attacks.

As shown in Table XI, we present detection performance comparison between PLUTO and baselines. Overall, compared with baselines, PLUTO improves AUC by an average of 1.83%, F1 Score by an average of 7.27%, and Recall by an average of 9.58%, and meanwhile, PLUTO reduces EER by an average of 27.96%.

Flow-based Attacker Filtering Setup: On one hand, we need to set S_{CW} to a value smaller than the burst duration of LDoS attacks. We consider that the burst duration is greater than the RTT, which means that the lower bound of burst duration is twice the bottleneck link delay (39.41 ms). Therefore, we set S_{CW} to 40 ms.

On the other hand, we assume an extreme case that a single burst send by attack flow contains at least one MTU. This means that the burst bytes is at least 1500 Bytes. As a result, we set BURST_TH to 1500.

Mitigation Performance Comparison: We compare the mitigation performance of PLUTO and baselines (P&F and NetBeacon) under the five traffic scenarios. As listed in Table IX (focusing on the column of Mitigation Testing), we use the ten groups of LDoS attack parameter to launch LDoS attacks. Each group of LDoS attacks is launched three times.

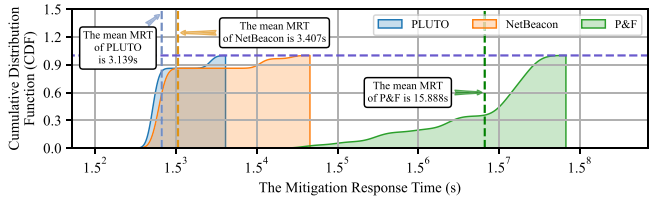


Fig. 29. The Measurement of Mitigation Response Time.

TABLE XII
RESOURCE CONSUMPTION PER PIPELINE IN PLUTO PROTOTYPE

TCAM (KB)	SRAM (MB)	Maximum Number of Parallel Operations in an MAU Stage	Maximum Number of Parallel Table Lookups in an MAU Stage
3.336 (0.053%)	1.895 (1.579%)	13 (5.804%)	4 (25.000%)

[1] We use blue to mark the proportion of PLUTO's resource consumption to the total resource of Intel Tofino. Here, TCAM is up to 6.2MB, SRAM is up to 120MB [20], number of parallel operations in an MAU Stage is up to 224, and number of parallel table lookups in an MAU Stage is up to 16 [49].

We measure the mitigation response time (MRT) for PLUTO and baselines. Concretely, MRT is the duration from the time LDoS attacks are launched to the time network returns to normal.

As shown in Fig. 29, we present the CDF with respect to MRT. Compared with P&F, the average MRT of PLUTO is significantly faster by 12.749 s, which benefits from the line-speed execution capacity of PLUTO. In addition, the data plane-aware design included in PLUTO, that is, activating TPSM for the Flow-based Attacker Filtering based on pre-inference results does not delay the attack mitigation compared with the traditional per-flow ML inference, i.e., NetBeacon. In particular, the average MRT of PLUTO is 0.268 s faster compared with NetBeacon, which benefits from the robust feature engineering for pre-inference.

Resource Consumption: As shown in Table XII, we measure the resource consumption of PLUTO, including the memory (TCAM and SRAM) usage, as well as the maximum number of parallel operations and table lookups in an MAU Stage. Measurement results indicate that PLUTO is resource-friendly to the P4 hardware switch, i.e., the Intel Tofino.

VIII. LIMITATION AND DISCUSSION

A. Occupation of Loopback Port Bandwidth

Since the P4 hardware switch, i.e., Intel Tofino has limited MAUs, PLUTO mirrors the packets and directs them to the loopback port, executing the rest process of LDoS attack detection. In particular, PLUTO outputs a mirrored packet to the loopback port only when a DW is completed, and each DW lasts the time of ($S_{SW} \cdot S_{DW}$). Meanwhile, referring to Section VII-B, the minimum S_{SW} and S_{DW} set by PLUTO are 1 ms and 2^3 respectively. Therefore, PLUTO delivers packets to the loopback port at a maximum packet rate of 125 packets/s. Assuming that the size of each mirrored packet is MTU, i.e., 1500 Bytes, the bandwidth of the loopback port occupied by PLUTO is 1.5 Mbps. This is much smaller than the bandwidth of a single port in Intel Tofino (100 Gbps). Therefore, PLUTO does not let the loopback port become a bottleneck.

B. Resource Redundancy in Function Operations

Within the P4 switch, a single MAU Stage can only be accessed once and the resources between different MAU Stages are isolated, thus the P4 Function Tool cannot reuse the tables and registers created for function operations. Each function operation has its independent tables and registers, which results in resource redundancy. Fortunately, referring to Section VII-A, the P4 Function Tool has a lightweight memory footprint. In Intel Tofino, a single function operation uses no more than 0.02% and 0.053% of the total TCAM and SRAM, respectively.

C. Scalability Analysis

The data plane-aware design proposed by PLUTO is a generic P4 design, containing the window-based pre-inference strategy and the time-limited per-flow state management. This design can be applied to a wide range of low-rate security threat solutions which is built through the resource-constrained P4 switch. In addition, although the P4 Function Tool only provides five basic function operations, their combination can achieve more complex operations. In Section VI-B2, we use the Logarithm and Exponent supported by the P4 Function Tool to compute multiplication, division, and power functions in the P4 switch.

IX. Conclusion

In this paper, based on the advantage of P4, we present PLUTO, a data plane-aware LDoS attack defense system built upon the P4 switch, defending LDoS attack at line speed. Within the data plane-aware design of PLUTO, we first propose the time window-based pre-inference strategy. We only maintain one group of states relevant to the aggregate flow for detecting LDoS attack at a macro level, thus the overhead incurred is significantly lightweight for the P4 switch. Besides, to further reduce the flow scale which the P4 siwtch handles, we propose the time-limited per-flow state management for conducting the Flow-based Attacker Filtering only when the pre-inference results indicates an LDoS attack occurs.

Furthermore, to practically deploy PLUTO on the P4 switch, we implement three modules: the P4 Function Tool, the Pre-inference Model Mapping, and the Flow-based Attacker Filtering. Here, the P4 Function Tool utilizes the scope reduction, achieving common function operations to compute extensive features in the P4 switch. The Pre-inference Model Mapping adopts an encoding-based mapping methods to deploy the pre-inference model on the P4 switch. In addition, the Flow-based Attacker Filtering leverages a P4-based deterministic data structure, i.e., the async-update hash table, to efficiently filter LDoS attack flows in the per-packet processing mode of the P4 switch. Compared with the baseline, we evaluate PLUTO from two aspects: 1) the accuracy and memory usage of the P4 Function Tool, and 2) the detection and mitigation performance. In future work, we will further optimize the details of PLUTO to make it adaptable to more security threats.

References

[1] Websites in Iran shut down due to LDoS attacks, 2009. [Online]. Available: https://www.okta.com/identity-101/slowloris/

[2] Government and institution websites in Italy down for at least one hour due to LDoS attacks, 2022. [Online]. Available: https://www.bleepingcomputer.com/news/security/italian-cert-hacktivists-hit-govt-sites-in-slow-http-ddos-attacks/

[3] D. Tang, X. Wang, X. Li, P. Vijayakumar, and N. Kumar, "AKN-FGD: Adaptive Kohonen network based fine-grained detection of LDoS attacks," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 1, pp. 273–287, Jan./Feb. 2023.

[4] 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.

[5] D. Tang, S. Zhang, Y. Yan, J. Chen, and Z. Qin, "Real-time detection and mitigation of LDoS attacks in the SDN using the HGB-FP algorithm," *IEEE Trans. Services Comput.*, vol. 15, no. 6, pp. 3471–3484, Nov./Dec. 2022.

[6] D. Tang, Y. Yan, S. Zhang, J. Chen, and Z. Qin, "Performance and features: Mitigating the low-rate TCP-targeted DoS attack via SDN," *IEEE J. Sel. Areas Commun.*, vol. 40, no. 1, pp. 428–444, Jan. 2022.

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

[8] D. Tang, S. Wang, B. Liu, W. Jin, and J. Zhang, "GASF-IPP: Detection and mitigation of LDoS attack in SDN," *IEEE Trans. Services Comput.*, vol. 16, no. 5, pp. 3373–3384, Sep./Oct. 2023.

[9] D. Tang, Z. Zheng, X. Wang, S. Xiao, and Q. Yang, "PeakSAX: Real-time monitoring and mitigation system for LDoS attack in SDN," *IEEE Trans. Netw. Service Manag.*, vol. 20, no. 3, pp. 3686–3698, Sep. 2023.

[10] C. Zheng and N. Zilberman, "Planter: Seeding trees within switches," in *Proc. SIGCOMM 2021 Poster Demo Sessions*, 2021, pp. 12–14.

[11] S. Ha, I. Rhee, and L. Xu, "CUBIC: A new TCP-friendly high-speed TCP variant," *ACM SIGOPS Operating Syst. Rev.*, vol. 42, no. 5, pp. 64–74, 2008.

[12] A. Kuzmanovic and E. W. Knightly, "Low-rate TCP-targeted denial of service attacks: The shrew vs. the mice and elephants," in *Proc. ACM SIGCOMM Conf.*, 2003, pp. 75–86.

[13] RFC 2988, 2000. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2988

[14] P. Bosshart et al., "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

[15] Public Tofino native architecture, 2021. [Online]. Available: https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf

[16] Arista 7170 multi-function programmable networking, 2020. [Online]. Available: https://www.arista.com/assets/data/pdf/Whitepapers/7170_White_Paper.pdf

[17] Behavioral model, (n.d.). [Online]. Available: https://github.com/p4lang/behavioral-model

[18] 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. 2022 IEEE Conf. Comput. Commun.*, 2022, pp. 1938–1947.

[19] 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.*, 2022, pp. 1099–1114.

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

[21] A. T.-J. Akem, M. Gucciardo, and M. Fiore, "Flowrest: Practical flow-level inference in programmable switches with random forests," in *Proc. 2023 IEEE Conf. Comput. Commun.*, 2023, pp. 1–10.

[22] B. M. Xavier, R. S. Guimarães, G. Comarela, and M. Martinello, "Programmable switches for in-networking classification," in *Proc. 2021 IEEE Conf. Comput. Commun.*, 2021, pp. 1–10.

[23] D. Barradas, N. Santos, L. Rodrigues, S. Signorello, F. M. 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.

[24] J. Yan et al., "Brain-on-Switch: Towards advanced intelligent network data plane via NN-driven traffic analysis at line-speed," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2024, pp. 419–440.

[25] M. Zhang et al., "Poseidon: Mitigating volumetric DDoS attacks with programmable switches," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–18.

[26] D. Ding, M. Savi, and D. Siracusa, "Tracking normalized network traffic entropy to detect DDoS attacks in P4," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 6, pp. 4019–4031, Nov./Dec. 2022.

[27] A. da Silveira Ilha, Â. C. Lapolli, J. A. Marques, and L. P. Gaspary, "Euclid: A fully in-network, P4-based approach for real-time DDoS attack detection and mitigation," *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 3, pp. 3121–3139, Sep. 2021.

[28] 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.
[29] A. G. Alcoz, M. Strohmeier, V. Lenders, and L. Vanbever, "Aggregate-based congestion control for pulse-wave DDoS defense," in *Proc. ACM SIGCOMM Conf.*, 2022, pp. 693–706.
[30] S. Kim, C. Jung, R. Jang, D. Mohaisen, and D. Nyang, "A robust counting sketch for data plane intrusion detection," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2023, pp. 1–17.
[31] H. Zhou, S. Hong, Y. Liu, X. Luo, W. Li, and G. Gu, "Mew: Enabling large-scale and dynamic link-flooding defenses on programmable switches," in *Proc. IEEE Symp. Secur. Privacy*, 2022, pp. 1625–1639.
[32] A. AlSabeh, E. Kfoury, J. Crichigno, and E. Bou-Harb, "P4DDPI: Securing P4-programmable data plane networks via DNS deep packet inspection," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2022, pp. 1–7.
[33] D. Tang, X. Wang, K. Li, C. Yin, W. Liang, and J. Zhang, "FAPM: A fake amplification phenomenon monitor to filter DRDoS attacks with P4 data plane," *IEEE Trans. Netw. Service Manag.*, vol. 21, no. 6, pp. 6703–6715, Dec. 2024.
[34] A. Laraba, J. François, S. R. Chowdhury, I. Chrisment, and R. Boutaba, "Mitigating TCP protocol misuse with programmable data planes," *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 1, pp. 760–774, Mar. 2021.
[35] M. Zhang et al., "NetHCF: Filtering spoofed IP traffic with programmable switches," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 2, pp. 1641–1655, Mar./Apr. 2023.
[36] D. Ding, M. Savi, and D. Siracusa, "Estimating logarithmic and exponential functions to track network traffic entropy in P4," in *Proc. IEEE/IFIP Netw. Operations Manage. Symp.*, 2020, pp. 1–9.
[37] N. K. Sharma, A. Kaufmann, T. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the power of flexible packet processing for network resource allocation," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 67–82.
[38] Y. Yuan et al., "Unlocking the power of inline floating-point operations on programmable switches," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2022, pp. 683–700.
[39] H. Namkung, Z. Liu, D. Kim, V. Sekar, and P. Steenkiste, "SketchLib: Enabling efficient sketch-based monitoring on programmable switches," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2022, pp. 743–759.
[40] G. Li et al., "IMap: Fast and scalable in-network scanning with programmable switches," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2022, pp. 667–681.
[41] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford, "BeauCoup: Answering many network traffic queries, one memory update at a time," in *Proc. ACM SIGCOMM Conf.*, 2020, pp. 226–239.
[42] V. Shrivastav, "Programmable multi-dimensional table filters for line rate network functions," in *Proc. ACM SIGCOMM Conf.*, 2022, pp. 649–662.
[43] K. Zhang, D. Zhuo, and A. Krishnamurthy, "Gallium: Automated software middlebox offloading to programmable switches," in *Proc. ACM SIGCOMM Conf.*, 2020, pp. 283–295.
[44] MAWI working group traffic archive, 2022. [Online]. Available: http://mawi.wide.ad.jp/mawi/
[45] Ryu SDN controller, 2017. [Online]. Available: https://github.com/faucetsdn/ryu/
[46] Mininet, 2022. [Online]. Available: http://mininet.org/
[47] D. Tang, Y. Yan, C. Gao, W. Liang, and W. Jin, "LtRFT: Mitigate the low-rate data plane DDoS attack with learning-to-rank enabled flow tables," *IEEE Trans. Inf. Forensics Security*, vol. 18, pp. 3143–3157, 2023.
[48] The Internet topology zoo, 2013. [Online]. Available: http://www.topology-zoo.org/index.html
[49] Tofino feature summary, 2021. [Online]. Available: https://opennetworking.org/wp-content/uploads/2021/05/2021-P4-WS-Vladimir-Gurevich-Slides.pdf

Dan Tang received the BS, MS, and PhD degrees from the Huazhong University of Science and Technology, in 2014. He is now an associate professor with the College of Computer Science and Electronic Engineering (CSEE), Hunan University (HNU). His research interests include network security, information security, and programmable network.

Boru Liu received the BS degree from the College of Computer Science and Electronic Engineering (CSEE), Hunan University. He is currently working toward the MS degree with CSEE, Hunan University. He is currently a senior with the College of Computer Science and Electronic Engineering (CSEE), Hunan University (HNU), Changsha, China. He is majoring in computer science and technology and his research focuses on programmable data plane and cyberspace security.

Keqin Li (Fellow, IEEE) is a SUNY distinguished professor of computer science with the State University of New York and also a national distinguished professor with Hunan University. His current research interests include cloud computing, fog computing and mobile edge computing, energy-efficiency computing and communication, embedded systems and cyber-physical systems, heterogeneous computing systems, Big Data computing, high-performance computing, CPU-GPU hybrid and cooperative computing, computer architectures and systems, computer networking, machine learning, intelligent, and soft computing.

Sheng Xiao received the PhD degree from the University of Massachusetts, Amherst, in 2013. He is an associate professor with the College of Computer Science and Electronic Engineering (CSEE) Hunan University (HNU), Changsha, China. His research interests include communication security, high performance computing, and data visualization.

Wei Liang received the PhD degree in computer science and technology from Hunan University, in 2013. He was a postdoctoral scholar with Lehigh University, Bethlehem, PA, USA, during 2014 to 2016. He is currently a professor with the School of Computer Science and Engineering, Hunan University of Science and Technology. His research interests include blockchain security technology, network security protection, embedded system and hardware IP protection, fog computing, and security management in wireless sensor networks.

Jiliang Zhang received the PhD degree in computer science and technology from Hunan University, Changsha, China, in 2015. From 2013 to 2014, he worked as a research scholar with the Maryland Embedded Systems and Hardware Security Lab, University of Maryland, College Park. He is currently a full professor with Hunan University. He is the director of Chip Security Institute of Hunan University, and the secretary-general of CCF Fault-Tolerant Computing Professional Committee. His current research interests include hardware security, integrated circuit design, and intelligent system.