Soter: Deep Learning Enhanced In-Network Attack Detection Based on Programmable Switches

Guorui Xie*†, Qing Li†¶, Chupeng Cui‡, Peican Zhu§, Dan Zhao†, Wanxin Shi*†,
Zhuyun Qi†, Yong Jiang*†, Xi Xiao*
* International Graduate School, Tsinghua University, Shenzhen, China
† Peng Cheng Laboratory, Shenzhen, China
¶ Southern University of Science and Technology, Shenzhen, China
‡ Jilin University, Changchun, China
§ Northwestern Polytechnical University, Xi'an, China

***Abstract*—Though several deep learning (DL) detectors have been proposed for the network attack detection and achieved high accuracy, they are computationally expensive and struggle to satisfy the real-time detection for high-speed networks. Recently, programmable switches exhibit a remarkable throughput efficiency on production networks, indicating a possible deployment of the timely detector. Therefore, we present Soter, a DL enhanced in-network framework for the accurate real-time detection. Soter consists of two phases. One is filtering packets by a rule-based decision tree running on the Tofino ASIC. The other is executing a well-designed lightweight neural network for the thorough inspection of the suspicious packets on the CPU. Experiments on the commodity switch demonstrate that Soter behaves stably in ten network scenarios of different traffic rates and fulfills per-flow detection in 0.03s. Moreover, Soter naturally adapts to the distributed deployment among multiple switches, guaranteeing a higher total throughput for large data centers and cloud networks.**

***Index Terms*—In-network intelligence, deep learning, network security.**

I. Introduction

Although protecting the network from attacks has been studied for a long time [1], [2], we have seen an increasing trend of network attacks for these years. A reason is that the cost of attacks can be cheap as the widely deployed but not well-defended IoT devices are easily hijacked for distributed criminal activities, challenging the detection efficiency [3]. Also, with the new emerging applications (e.g., smart housing and e-health), the generated mixed traffic on the network is complex for the traditional attack detection schemes like signature-based detection, degrading the detection accuracy [4].

Recently, deep learning (DL) has been applied to the complex traffic pattern and achieved a superior performance [5]–[8]. For example, in [6], the authors propose 1DCNN and LSTM models for attack detection. In [7], the authors propose detectors with hidden neural layers of one or three. Their experiments confirm that the DL models outperform the conventional machine learning methods, e.g., the Decision Tree (DT) [9] and the Logistic Regression (LR) [10]. Though many DL-based solutions reach better accuracy and defeat their predecessors, they are computation-intensive and have to be conducted on the specific GPU-equipped servers instead of in-network devices (e.g., switches) with limited performance [11], [12]. In addition, even with computation acceleration by GPUs, these models typically run in the order of milliseconds, which can not provide a satisfactory throughput for large-scale networks (e.g., data centers with tens of thousands of entities and the traffic rate of 100Gbps) [13].

Given the high throughput and capability of customized packet processing, programmable switches (e.g., P4 switches [14]) have been deployed in commercial data centers and cloud service networks [15], [16]. The P4 switch's data plane consists of a match-action pipeline for high-speed (e.g., hundreds of Gbps) packet processing. Network administrators can define the rules and actions for packet matching in different tables and then install them on the pipeline. Hence, several works have attempted to offload the learning-based detection/classification models to the switches' data plane for the real-time processing, i.e., in-network intelligence [17]–[22]. Although P4 switches provide a high throughput for in-network intelligence solutions, they only support limited actions for packet processing on the data plane. Floating-point mathematical operations such as multiplication and division are not supported [23], [24]. As such, existing P4 switch based solutions mostly adopt the rule-based machine learning classifier, i.e., the DT, rather than the more accurate DL models which require floating-point operations.

In this paper, we present Soter[1], a deep learning enhanced in-network approach for fast and accurate network attack detection. Soter leverages the power of DL to build an accurate attack detector on the in-network switches. Furthermore, as switches are responsible for the traffic of different subnetworks, Soter can act as a distributed detection system, achieving a high global throughput in the whole network scope. However, to achieve this high-level design goal, a few challenges need to be addressed: first, how to deploy a DL model on the switch as its floating-point operations are not supported in the switch's data plane, not to mention that the

Corresponding author: Qing Li (liq@pcl.ac.cn)

[1]In Greek mythology, Soter is the deity of security and protection.

performance of the switch is lower than that of GPU-enabled servers; second, how to run Soter in a timely manner in switches when facing large networks (e.g., data centers and cloud networks), because these switches usually have a traffic rate of tens or hundreds of Gbps.

We address these challenges with the following key ideas:

- We propose a lightweight deep neural network, called Branch Convolution Net (BCN), for programmable switches. Following the divide-and-conquer paradigm, we divide a convolution operation into several efficient branch convolutions, significantly reducing the original complexity and neural parameters. In this way, this neural network can be run on switches' CPUs at a speed of $\sim 0.06ms$ per detection without the GPU acceleration.
- We design an intensive detection module, with several BCNs together, installed on the switch CPU to speed up the attack detection. The intensive detection module makes the best use of the CPU multi-core and memory to run the traffic buffering, processing, and detection in a parallel fashion.
- We design a two-phase detection scheme, where a DT-based P4 program on the data plane first quickly filters massive benign packets according to packet-level features and only clones the suspicious packets to the CPU; then, the intensive detection module on the CPU conducts a thorough in-depth investigation on the suspicious packets using flow-level features to produce accurate detection results.

We implement a prototype of Soter on a commodity switch[2]. To the best of our knowledge, this is the first attempt to explore the deep learning-based attack detection on P4 switches. Thorough experiments reveal that: **1)** Compared with the state-of-the-art works [5], [6], [8], [25], [26], the BCN incurs 193x fewer parameters and 165x fewer floating-point operations. With such a lightweight model, the BCN still achieves a high accuracy of 98.25%. **2)** Soter runs stably under ten network scenarios with various bit rates and flow rates. Especially, with a bit rate of 100Gbps and a flow rate of 16Kfps, the detection time of a flow that goes through Soter is $0.03s$, which is promising for large-scale networks.

II. Background

A. DL-based Attack Detection Solutions

DL algorithms have long been used to detect network attacks. In [5], the authors present three CNN models with different neural layers: shallow CNN, moderate CNN, and deep CNN, with one, two, and three convolution layers after the input layer, respectively to study the relationship between the model depth and the performance. In [6], the authors design a combined 1DCNN+LSTM model to boost the detection performance. That is, an LSTM module is after a 1D convolution and max pooling layer. At the end of the module, an extra fully-connected and softmax layer are added for the classification. The experimental results on the public dataset of UNSW-NB15 [27] demonstrate the superior performance of 1DCNN+LSTM when compared with the machine learning-based random forest and support vector machine models. In [7], the authors utilize different numbers of the fully-connected layers to build two models: ANN with one fully-connected layer and DNN with three fully-connected layers. Both models are also trained and evaluated on the UNSW-NB15 dataset. The comparison results show that ANN and DNN outperform conventional methods, including DT and LR. In [8], a 2DCNN with two 2D convolution layers is proposed for attack classification. In 2DCNN, the bytes of each labeled flow are converted to a 2D gray image for 2DCNN training or evaluation. Other DL-based detection approaches are proposed in [25], [26], [28]–[31].

Though these DL-based works have outperformed the precedent non-DL solutions and demonstrated considerable promise, they require powerful hardware (e.g., GPU-equipped X86 servers), which hinders their applications on in-network devices [11], [12]. As a result, these solutions must be performed in a centralized way. Typically, the traffic is periodically collected on in-network monitors and then sent to the central server for analysis, which may cost a long time for attack responses and cause the potential financial loss.

B. P4 Switch And Its Constraints

Compared with the high-performance X86 servers, P4 switches support up to Tbps of throughput while allowing custom packet processing, but at the expense of computational capability [14]. P4 switches contain two types of processors. One is a weak embedded CPU, which is responsible for the local data plane management. For instance, network administrators can utilize it for P4 program compilation, debugging, and installation. It can also be used to configure the P4 match-action rules, e.g., adding or removing rules in a running P4 program on the data plane. The other processor is the Tofino ASIC [32], acting as the switch's data plane and running the customized P4 program. The architecture of the Tofino ASIC mainly contains a parser, a match-action pipeline, and a deparser. Incoming packets are first mapped into packet header vectors (PHV) by the parser. A PHV contains the parsed IP addresses, IP flags, TCP/UDP ports, etc., for a packet. Then, these PHVs are passed to the pipeline stage by stage. During each pipeline stage, one or several fields of the PHV may match (M) a given table rule, and then trigger the associated action (A) for the further processing.

Thanks to the well-designed characteristics, several production data centers and cloud service networks have deployed P4 switches to take advantage of their efficiency benefits [15], [16]. However, only simple operations like integer additions and bit shifts are allowed in P4 actions. Thus, it is hard to implement DL-based detection approaches, which require floating-point mathematical operations, on the Tofino ASIC.

C. Learning on P4 Switches

In contrast to complicated DL models, the DT is a rule-based machine learning model that requires no floating-point

[2]Codes are available at https://github.com/xgr19/Soter

operations during the inference. Therefore, recent works are proposed to completely offload the DT on the Tofino ASIC for real-time networking classification and detection, i.e., in-network intelligence [17]–[22].

In [17], the authors propose a framework, IIsy, for mapping the DT into an IoT traffic classification P4 program. In IIsy, the number of tables in the P4 program is equal to the number of fields in a PHV used as the DT input plus one. Each table matches a specific feature with all its potential values. The matched value is assigned to a metadata field in the PHV, indicating the branch to take in the DT. In the last table, the metadata fields are used as keys to finally output the matched classification results for the packet represented by the PHV. In [18], the authors propose SwitchTree for the DT-P4 conversion. SwitchTree maps every DT level to a single P4 table. Once a PHV passes the P4 program, the matched results of the previous table are encoded into the metadata fields in the PHV and act as the match key for the next table. In [21], the authors deploy a DT for the intrusion detection. They first traverse the DT from the root to each leaf, extracting the corresponding rules. Then, these rules are hard-coded into the *if-else* P4 statements for detection purposes. In [19], [20], the authors also develop P4 programs to speed up traditional machine learning classifications based on DTs.

Though benefiting from the high throughput of P4 switches, the current in-network solutions' learning detectors (i.e., DTs) are weaker than DL-based ones. The rule-based structure guarantees their feasibility on Tofino ASIC but also limits their detection capability [6], [7]. Thus, in this paper, we seek to explore a learning solution for in-network attack detection that hits the sweet spot—feasible for deployment on P4 switches to ensure high processing throughput, but still achieves satisfying model accuracy.

III. Soter Overview

Problem statement. Soter detects attack traffic by taking IP packets as input. First, it filters potentially malicious packets by packet headers (eg, TTL, IP flags). The filtered packets are then organized in flows by their 5-tuples for the second round of detection based on flow-level features (e.g., sum/average/variance of packet lengths). Finally, Soter outputs detected malicious 5-tuples for packet blocking. Based on this, we depict Soter, which consists of two main processes (the training process and the detection process), in Fig. 1.

Training process. In Soter, we need a high-performance central controller to take on the responsibilities of the computationally expensive multi-model (DT and BCN) training and massive training data storage. During the training, the **feature selection** first extracts examples of the packet-level and flow-level (identified by the 5-tuple) features from the traffic data of benign flows and different attacks. Then, the examples of packet-level features are used for the DT training. While the examples of flow-level features are used to train the proposed **Branch Convolution Net (BCN)**. With well-designed branch convolutions, the BCN reduces the computation resource but still retains a competitive accuracy. After the training process,

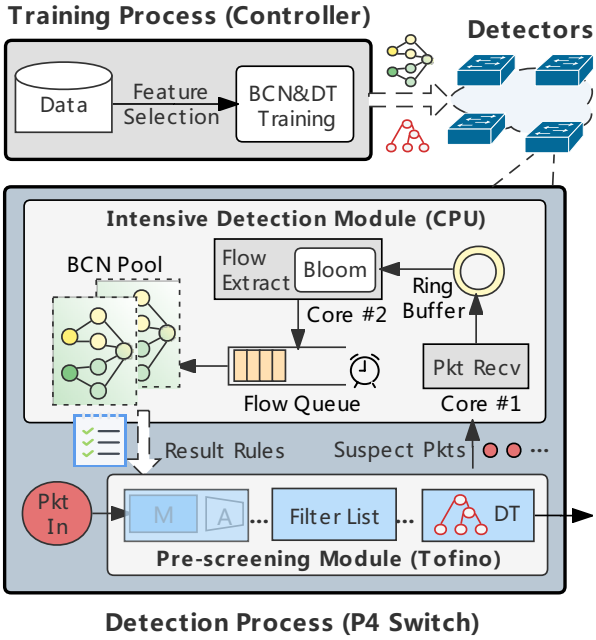


Fig. 1: The Soter framework.

a copy of the (BCN, DT) model pair is sent to each P4 switch in the network for conducting the following two-phase detection process.

Detection process. In a P4 switch, the **pre-screening module** in the Tofino ASIC is a P4 program consisting of two parts: the first is a filter list, if a packet matches (M) the illegal 5-tuples in the list, the triggered action (A) will discard this packet; if a packet matches the legal 5-tuples, the triggered action will send it to the switch egress without considering the subsequent detection; otherwise, the packet will go to the second part of our P4 program, i.e., the trained DT model in the P4 form. The responsibility of the second part is to filter the suspicious packets of attacks and clone them to the intensive detection module for the BCN inspection so as to avoid the comprehensive DL detection on the large-volume traffic. The **intensive detection module** on the CPU is delicately designed to leverage the BCN for a more accurate detection of suspicious packets. In detail, we receive suspicious packets at core#1 and write them in the ring buffer. Meanwhile, the core#2 runs the flow-level feature extraction process for the packets (the packets of detected flows will be removed by the bloom filter [33]). In addition, a BCN pool maintains several BCN instances on the other cores. For the batched flows in the flow queue, if the needed features are ready or the waiting time is out, the idle BCN model is chosen to detect these flows. After the detection, the result rules recording all the legal and illegal 5-tuples are sent to the filter list.

In this two-phase way, only a small volume of the packets are sent to the BCN. The intuition behind Soter is that the distributed attack across the network does not generate large malicious traffic in a subnetwork charged by a switch. Moreover, if an attack starts just now, its traffic is relatively small,

and Soter can block it immediately. Furthermore, with Soter, we can build a distributed attack detection across multiple in-network switches, covering the whole large-scale network.

IV. Feature Selection

Feature selection plays a significant role in learning algorithms [34], [35]. As such, in this section, we discuss the input packet-level and flow-level features used in the DT and the BCN, respectively.

A. Packet-level features for the DT

Recently, several methods are proposed to collect and calculate features such as the average packet size and total packet count of a flow on the Tofino ASIC [18], [20], [23]. Though these flow-level features are informative for the detection, their implementation has several limitations. First, as the Tofino ASIC does not support floating-point multiplication and division, they have to estimate these features, resulting in inaccurate values. Second, their estimation consumes too much memory because multiple registers must be allocated to maintain the states of each ongoing flow. Based on this, we decide to simply utilize the packet header fields in the PHV for the DT, leaving the complex flow-level computation to the CPU module. Nevertheless, not all fields in headers are taken into account. For example, the checksum, identification, and fragment offset are vital for forwarding the packets between the switches but are uninformative for our detection [36]. Also, the IP addresses and TCP/UDP ports are strongly related to the subnetwork configuration, which hinders the generalizability of DT [13], [37]. At last, we select the following six header fields in the PHV: the header length, the IP protocol, the type of service, the time to live, the IP flags, and the TCP flags (zeros for the UDP packets).

B. Flow-level features for the BCN

In contrast to the Tofino ASIC, the switch CPU allows us to flexibly calculate the complex flow-level features, which is desired for a precise detection. In detail, we arrange the packets with the same 5-tuple (i.e., the source IP/port, destination IP/port, and protocol) together and compute features like the sum/average/variance of the packet lengths. However, waiting for the all packets' arrival is time-consuming, which violates the intention of detecting an ongoing attack in time. Therefore, we design a novel neural network, i.e., BCN, which only uses the sum/average/variance length of the first three packets and the original 252 payload bytes of a flow during the detection.

V. Branch Convolution Net

Up to now, convolution operations, especially the 1D convolution, play an essential part in the attack detection [5], [6], [8], [31]. Nevertheless, little of these works discuss the model compression for the limited-performance P4 switches. Thus, in this section, we first discuss the limitations of the classical 1D convolution. Then we propose the lightweight 1D branch convolution operation, and finally build the branch convolution net (BCN) based on it.

A. One-Dimensional Convolution

For a classical 1D convolution, its procedures are described as follows. Let $z^{in} \in \mathbb{R}^{N \times C}$ denote an input of length N and the number of channels is C. A convolution kernel $k \in \mathbb{R}^{W \times C}$ will filter a window of W input per time and combine the C channels of each filtered input together to produce an output $z \in \mathbb{R}^{N'}$:

$$z_x = \sum_{i=1}^{W} \sum_{j=1}^{C} k_{i,j} \times z^{in}_{x+i,j}, \tag{1}$$

where $N' = N - W + 1$ and $x \in [1, N']$. The final output of a convolution layer with M kernels is $z^{out} \in \mathbb{R}^{N' \times M}$ where $z^{out} = \{z^1, \ldots, z^M\}$ is the set of the M kernel outputs. Accordingly, the number of floating-point parameters used in this convolution layer P_{conv} is:

$$P_{conv} = M \times W \times C, \tag{2}$$

indicating the cost depends multiplicatively on the number of kernels, the size of the kernel window, and the number of the input channels. Apparently, this parametric cost is linearly related to the memory and the floating-point operations (e.g., the multiplication and sum).

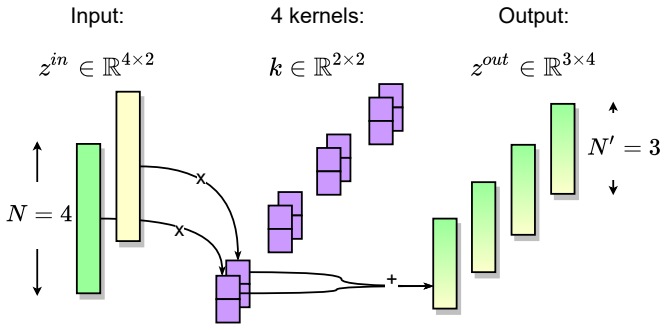


Fig. 2: The 1D convolution layer.

Fig. 2 depicts an example of a 1D convolution layer. As shown, the input $z^{in} \in \mathbb{R}^{4\times2}$ has a length of $N = 4$ and the number of channels is $C = 2$. To produce an output $z^{out} \in \mathbb{R}^{3\times4}$ with four channels, this example layer consists of four kernels (i.e., $M = 4$) where each kernel $k \in \mathbb{R}^{2\times2}$ has a filter window of $W = 2$ and the number of channels is $C = 2$ (exactly the same as z^{in}). As such, this example layer introduces $M \times W \times C = 4 \times 2 \times 2 = 16$ floating-point parameters. However, a neural network usually has tens of thousands of parameters per convolution layer.

B. Branch Convolution

Though there exist other model simplifications in DL (e.g., MobileNet [12], ShuffleNet [11]), they are mainly used for computer vision (CV). Conditions of CV and networking are different: CV models use 2D-CNNs but networking prefers 1D-CNNs, and CV models are more complex (MobileNet/ShuffleNet has ∼500M FLOPs which is much higher than many networking models) Therefore, we present our simplification, the branch convolution, for network detection.

At first, we divide the input $z^{in} \in \mathbb{R}^{N \times C}$ into C branches. Each branch only takes charge of the elements in its assigned channel from z^{in}, i.e., $z^{in,b} \in \mathbb{R}^{N}$ where b is the index of the assigned channel. Then, M/C kernels are assigned to each branch for the convolution operations. A branch convolution kernel is in the form of $k \in \mathbb{R}^{W}$. The new convoluted output $z^{b} \in \mathbb{R}^{N'}$ is given as:

$$z_x^b = \sum_{i=1}^{W} k_i \times z_{x+i}^{in,b}, \tag{3}$$

where $N' = N - W + 1$ and $x \in [1, N']$. After the convolutions, a branch b has an output of $z^{out,b} \in \mathbb{R}^{N' \times M/C}$ which is the set of $\{z^{b,1}, \ldots, z^{b,M/C}\}$. Finally, we merge the different branch outputs (i.e., $z^{out,b}$) together to form the final output $z^{out} \in \mathbb{R}^{N' \times M}$. As a result, the number of floating-point parameters used in the branch convolution layer P_{branch} is:

$$P_{branch} = M \times W. \tag{4}$$

By replacing the classical one-dimensional convolution of the branch convolution, we achieve a parametric cost reduction of:

$$\frac{P_{branch}}{P_{conv}} = \frac{M \times W}{M \times W \times C} = \frac{1}{C}. \tag{5}$$

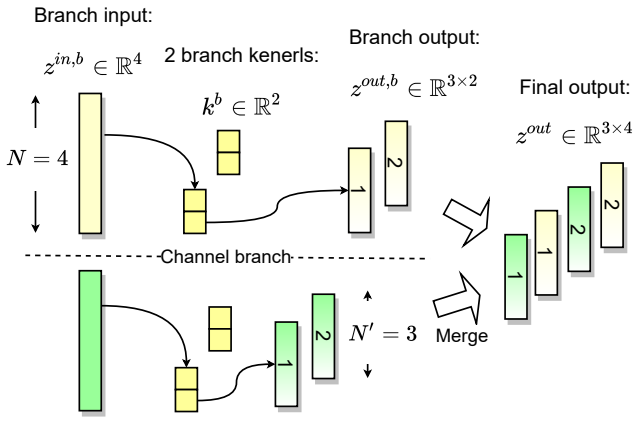


Fig. 3: The branch convolution layer.

Fig. 3 demonstrates an example of a branch convolution layer. As shown, the input consisting of $C = 2$ channels is divided into two branches. For the top branch, its input $z^{in,b} \in \mathbb{R}^4$ has a length of $N = 4$. To get a final output of $M = 4$ channels, this branch is assigned with $M/C = 2$ kernels where each kernel $k^b \in \mathbb{R}^2$ has a filter window of $W = 2$. After the convolutions of two kernels, we have the branch output of $z^{out,b} \in \mathbb{R}^{3\times2}$. At last, the output of the top and bottom branch is merged to reach the final output of $z^{out} \in \mathbb{R}^{3\times4}$. This example layer incurs $M \times W = 4 \times 2 = 8$ floating-point parameters, which is half of the 1D convolution layer in Fig. 2.

Although the branch convolution reduces the parametric cost by C times, a side effect of branching the input along the channels is that it may prevent the kernel from exploring the inter-channel information, weakening the learned representation and accuracy. Therefore, in the BCN, we consider the 1D convolution and branch convolution jointly.

C. BCN Architecture

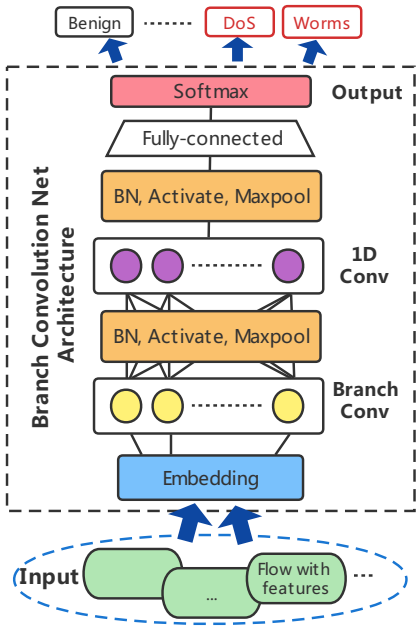


Fig. 4: The BCN architecture.

The architecture of the BCN is illustrated in Fig. 4. At first, the flow-level feature $f \in \mathbb{R}^{N}$ of a flow is fed to the embedding layer. The embedding layer is useful to increase the channels of each input feature and enrich the learned representation [38]. After the embedding layer, we obtain an output/representation $f \in \mathbb{R}^{N \times C}$ where C is the channels generated by the embedding. Next, this representation is divided into C branches, passing the branch convolution layer with M_1 kernels and generating the new representation of $f \in \mathbb{R}^{N_1 \times M_1}$.

Before going to the 1D convolution layer, the representation is fed to the batch normalization (BN), activation function (Activate), and max pooling (Maxpool) layers in order. BN [39] normalizes the layer input representation and helps make training faster and more stable. After the BN, an activation function called hard-swish [40] is used to enrich the nonlinearity of the representation. Then, the max pooling is conducted to down-sample the representation, reducing its dimensionality [41]. Next, we apply a 1D convolution layer with M_2 kernels, reaching a representation of $f \in \mathbb{R}^{N2 \times M_2}$.

Finally, this representation is passed to the fully-connected layer and the softmax layer in order. For a classification with K classes (e.g., the benign and different attacks), the fully-connected layer maps the input into the K-dimensional space, generating new $f \in \mathbb{R}^{K}$. Then, the softmax layer outputs the predicted probabilities of the K classes and selects the class label with the maximum probability as the output. The detailed

parameter settings of the BCN are discussed in the experiments in Section VIII-A.

VI. Intensive Detection Module

In this section, we present the intensive detection module installed on the switch CPU to coordinate multiple tasks (e.g., the packet receiving, the flow feature extraction, the BCNs) involving the detection and achieve a higher quality of the detection. The intensive detection module mainly considers two aspects, i.e., the multi-core processing and the packet/flow buffering.

A. Multi-core Processing

CPUs are now equipped with multi-core technology that allows the system to perform multiple tasks concurrently for higher overall system performance. As illustrated in Fig. 1, we spread the tasks such as packet receiving (Pkt Recv), flow-level feature extraction (Flow Extract), and BCN across different cores for the acceleration. In particular, core#1 is assigned to run the process of receiving potentially malicious packets. Core#2 fetches the packets, reorganizes them into flows, and then extracts the flow-level features used by the BCN. All other cores form a BCN pool, each hosting a BCN to process the extracted flows.

B. Packet and Flow Buffering

Though the multi-core processing improves the overall detection speed, the synchronization among the multiple processes is not easily achieved. Typically in the high-speed network, packets arrive at a much faster speed than both the flow-level feature extraction and the BCN detection. To handle the synchronization among these tasks, we utilize three data structures, i.e., the ring buffer, the bloom filter, and the flow queue, as shown in Fig. 1. Suspicious packets are first stored in the ring buffer and then the packets of detected 5-tuples are eliminated by the bloom filter (before the Flow Extract). Finally, the flow queue maintains the extracted flow-level features waiting for the BCN detection.

The **ring buffer** can be regarded as a fixed-size circular array with a head and a tail. The packet receiving process adds packets at the tail of the array, while the flow-level feature extraction process fetches packets from the head of the array. In this way, we can avoid the read–write conflict between the processes of packet receiving and flow feature extraction, reducing the cost of the inter-process communication. The **bloom filter** is a space-efficient data structure, which utilizes k independent hash functions, each associated with a m bit array, to support set membership queries [33]. To add an element (e.g., the 5-tuple), each hash function hashes the element, and the position in its corresponding bit array is set to 1. To test whether an element is in the set, the element is hashed by all hash functions. It is said to be in the set only if all corresponding positions in the bit arrays are ones. The **flow queue** is employed to store the extracted features of each flow and feed the features to the BCN models for the subsequent attack detection whenever a BCN becomes available. To improve the insertion efficiency, we apply two hash functions on the 5-tuple to locate the bucket of each flow, and the buckets are in the form of chains to resolve the hash collision. Moreover, we set a timer for each flow stored in the queue so as to force the timeout flows to be detected. The detailed settings of these data structures are discussed in Section VIII-A.

VII. Pre-screening Module

In this section, we present the DT-based P4 program to filter suspicious packets to the CPU for the BCN detection, further relieving the detection pressure of the intensive detection module. The P4 program consists of two main components, the filter list and the converted DT model.

A. Filter List

The P4 code of the filter list is shown in Listing 1. The filter list follows the match-action paradigm, which uses the 5-tuple of a packet as the key for the action match. By default, the filter list takes no action on passing packets. However, once the 5-tuple in a packet's PHV *exactly* matches a rule sent by the BCN, the $drop()$ or $set\_detected()$ action is called. The $drop()$ action sets the (*drop_ctl*) flag in the packet and forces it to *exit* the pipeline. Then, the deparser will discard this packet. The $set\_detected()$ action is applied on the packets belonging to benign flows detected by the BCN. It forces the packets to exit the following detection pipeline and go to the switch egress.

Listing 1: The filter list in P4 language

```
action drop(){
    ig_dprsr_md.drop_ctl = 0x1;
    exit;
}
action set_detected(){
    // a suspicious flow that has been detected to
        be benign by the BCN
    exit;
}

table filter_list{
    key = {
        hdr.ipv4.src_addr: exact; // exact match
        hdr.ipv4.dst_addr: exact;
        ...
    }
    actions = {drop; set_detected; NoAction;}
    const default_action = NoAction;
}
```

B. DT-P4 Conversion

Typically, a trained DT classifies a packet to be benign or one of the illegal classes as follows. A DT node compares a specific input feature of the packet with its *threshold* and then routes the packet to its left ($\leq threshold$) or right ($> threshold$) child in the next level. Starting from the root, this comparison is recursively repeated until a leaf node is reached. Finally, the class label of the leaf node is returned, which indicates the packet is benign or illegal.

In a P4 switch, a packet traverses multiple consecutive pipeline stages in the form of the PHV. Each stage can

read and modify the packet properties, which affects the processing in the next stage. Therefore, to host a DT, each stage represents a level of the DT, and the PHV conveys the input features. Listing 2 depicts one DT level in P4. The keys {*meta.prev_node_id, meta.comparison_flag*} represent the comparison result in the previous node, and the other keys (e.g., *hdr.ipv4.ttl, hdr.flag*) indicate the header fields used for comparison in this level. The $clone\_to\_cpu()$ action copies the packet associated with the matched PHV to the CPU for further attack analysis. The $check\_feature()$ action replaces the node id and the comparison result in the PHV with the input parameters of the current node rule. The $set\_benign()$ action means the packet associated with the matched PHV is benign and forces this packet to *exit* the following DT level comparisons in the pipeline.

Listing 2: A DT level in P4 language

```
action clone_to_cpu(){
    ig_tm_md.copy_to_cpu = 0x1;
    exit;
}
action check_feature(bit<8> node_id, bit<1>
    less_than_threshold){
    meta.prev_node_id = node_id;
    meta.comparison_flag = less_than_threshold;
}
action set_benign(){
    exit;
}
table level{
    key = {
        meta.prev_node_id: exact;
        meta.comparison_flag: exact;
        hdr.ipv4.ttl: range; // range match
        hdr.flag: range; // TCP flag (zero in UDP)
        ...
    }
    actions = {check_feature; set_benign;
        clone_to_cpu;}
}
```

Assume node#3 ($node\_id = 3$) of this level is the left child ($comparison\_flag = 1$) of the node#1 ($prev\_node\_id = 1$) in the previous level, and it utilizes the $ttl = 32$ for comparison. Then we will install a P4 rule:

```
add_with_check_feature(prev_node_id=1,
    comparison_flag=1, ipv4.ttl.start=0, ipv4.ttl.
    end=32, hdr.flag.start=0, hdr.flag.end=INF,
    node_id=3, less_than_threshold=1)
```

where a packet with $ttl < 32$ will match this rule and trigger the check_feature(node_id=3, less_than_threshold=1) action. Unlike the prevalent P4 virtual machine bmv2 [42] used in [17], [18], [20], our P4 program is designed to run on the hardware, which faces more critical constraints. For instance, the hardware switch does not support dynamic keys, so we have to declare all potential keys in advance, and set the irrelevant keys (e.g., the *hdr.flag* in the example) in a rule with the infinite range.

VIII. Evaluation

In this section, we first introduce the experiment settings. Then, the performance of the DT, BCN, and the overall two-phase detection is evaluated. Finally, we analyze the efficiency of the Soter prototype under the high-speed artificial traffic.

A. Experimental Setup

1) Data: To evaluate the detection accuracy, we utilize the public dataset of UNSW-NB15 [27], which is widely used in attack detection [6], [7]. Table I describes the statistics of the dataset captured in February 2015. As shown, the traffic of attacks has a small amount. Moreover, to evaluate the detection efficiency, we employ the traffic generator (KEYSIGHT XGS12-SDL[3]) to simulate high-speed traffic of 40Gbps and 100Gbps under the Internet Mix (IMIX) mode. The IMIX generates hybrid-length packets, and its performance is expected to approximate the reality [43], [44].

Table I: The UNSW-NB15 dataset.

Binary Class	Multi Class	Abbr.	Flows	Packets	Storage
Attack	Fuzzers	Fuz	15494	69511	49GB
	Analysis	Ana	313	800	
	Backdoor	Bac	241	784	
	DoS	DoS	2987	17698	
	Generic	Gen	3172	16112	
	Reconnaissance	Rec	9923	19894	
	Shellcode	She	1288	2576	
	Worms	Wor	139	320	
	Exploits	Exp	20621	152290	
Benign	Benign	Ben	875778	7728303	

2) Hardware: The DT, BCN, and other compared models are trained on the controller with a CPU of Intel(R) Xeon(R) Gold 6230R CPU @ 2.10GHz and a GPU of GTX 2080Ti. Also, we employ a commodity P4 switch of H3C S9850-32H[4] for the overall efficiency evaluation of the Soter prototype.

3) Model settings: On the controller, we build a DT of eight levels with the scikit-learn machine learning library. The DT is trained with the packet-level features suggested in Section IV-A. We implement the BCN with the widely used DL library Pytorch, the detailed settings of BCN are listed in Table II. On the H3C commodity switch, the trained DT is converted to a P4 program in the $P4_{16}$ language. The trained BCN is run with the MNN [45], a lightweight DL inference library.

Furthermore, other DL-based approaches including 1DCNN [25], 2DCNN+LSTM [26], 2DCNN [8], DCNN, MCNN, SCNN (all in [5]), and 1DCNN+LSTM [6] are reproduced by Pytorch for our comparison. Each scheme is evaluated with a five-fold cross validation. That is, the original dataset is randomly partitioned into five equal-sized subsets. Among the five subsets, a single subset is retained as the validation data. The remaining subsets are used as the training data. Then, this cross-validation process is repeated five times, with each of the five subsets used exactly once as the validation data.

[3]https://www.keysight.com/us/en/products/network-test/network-test-hardware/xgs12-chassis-platform.html?rd=1

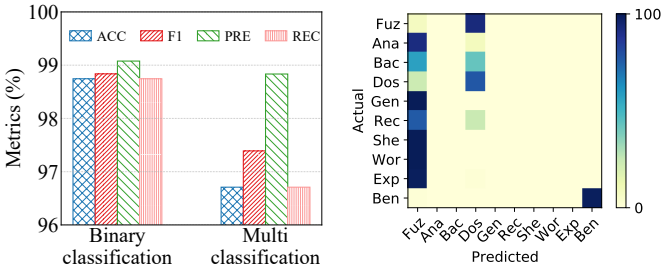
[4]https://www.h3c.com/en/Products_Technology/Enterprise_Products/Switches/Data_Center_Switches/H3C_S9850/

Table II: BCN implementation.

Layer	Description	Output
Input	Flow-level features in Section IV-B	$f \in \mathbb{R}^{258}$
Embedding	The number of chnnels $C = 16$	$f \in \mathbb{R}^{258\times16}$
Branch Conv	The number of kernels $M_1 = 64$, the filter window $W = 8$, stride 5	$f \in \mathbb{R}^{51\times64}$
BN, Activate	The hard-swish function	$f \in \mathbb{R}^{51\times64}$
Maxpool	The filter window $W = 3$, stride 3	$f \in \mathbb{R}^{17\times64}$
1D Conv	The number of kernels $M_2 = 64$, the filter window $W = 8$, stride 5	$f \in \mathbb{R}^{3\times64}$
BN, Activate	The hard-swish function	$f \in \mathbb{R}^{3\times64}$
Maxpool	The filter window $W = 4$, stride 3	$f \in \mathbb{R}^{1\times64}$
Fully-connect		$f \in \mathbb{R}^{10}$ (10 classes)
Softmax	Calculate class probabilities	Class label
Others	The batch size is 128, epoch is 5, optimizer is Adam with learning rate of 0.001	

4) Intensive detection module settings: The whole intensive detection module is built in C++. We use the MurmurHash [46] in both the bloom filter and the flow queue. In addition, the bloom filter utilizes 20 hash functions and 3MB memory to guarantee a hash collision probability below 10^{-6}. According to our analysis on the UNSW-NB15 dataset, $\sim 99\%$ flows send their first three packets in three seconds. Thus, we set the timeout of the flow queue to $3s$. This is also supported by the [47], [48] as their measurement shows that the 80% flows in data centers finish in $10s$.

B. DT Performance



(a) Detection Metrics (b) Confusion matrix

Fig. 5: The detection metrics of the binary/multi-class DT and the confusion matrix of the multi-class DT.

Fig. 5a shows the detection accuracy (ACC), F1-score (F1), precision (PRE), and recall (REC) of the DT on the UNSW-NB15 dataset. The binary classification refers to the detection of benign/malicious in the DT, and the multi-classification means that the DT detects whether the packet is benign or a particular type of attack. As shown, the DT performs better on the easier binary classification, reaching a higher ACC of $\sim 98.75\%$. In particular, the multi-class confusion matrix in Fig. 5b demonstrates that the DT incorrectly predicts most ($\sim 92\%$) of the malicious packets as the Fuz attack, indicating the DT is far from competent in distinguishing different types of attacks. However, the multi classification is usually more valuable. For example, it allows the administrators to react more precisely to an ongoing attack, and it is also helpful for the deep investigation of a specific attack. Therefore, in Soter, we use the DT to filter the illegitimate packets, leaving the complicated multi-class detection to the more powerful BCN.

C. BCN Performance

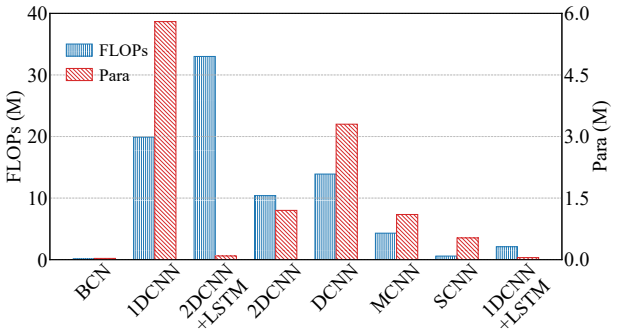


Fig. 6: The model complexity of different DL-based approaches.

Fig. 6 illustrates the model complexity of the BCN and other DL-based solutions. Here, we use the floating-point operations (FLOPs) and parameters (Para) to evaluate the model complexity. Thanks to the simpler branch convolution, our proposed BCN has the lowest FLOPs and Para. For instance, when compared with the 1DCNN, the BCN has a 165x lower FLOPs and a 193x less Para. When compared with the SCNN (another lightweight model), the FLOPs and Para of the BCN are also 5x and 18x lower, respectively. These results all imply the superior lightweight of the BCN.

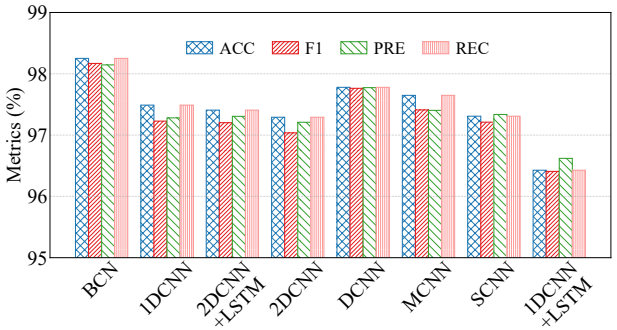
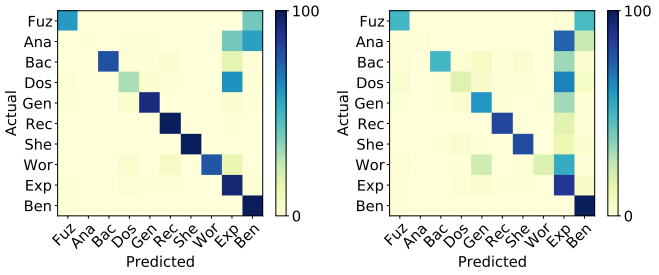


Fig. 7: The detection metrics of different DL-based approaches.

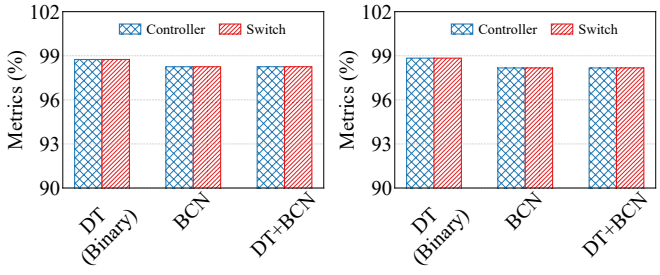
While such significant complexity reduction helps the BCN to run fast on the weak switch CPU (discuss later in Section VIII-E), one may worry that it would degrade the detection performance. However, the detection performance evaluation in terms of different metrics on the UNSW-NB15 dataset in Fig. 7 shows that the BCN outperforms other



(a) BCN (b) 2DCNN+LSTM

(c) DCNN (d) MCNN

Fig. 8: The confusion matrix of different DL-based approaches.



(a) ACC (b) F1

(c) PRE (d) REC

Fig. 9: The detection metrics of the binary-classs DT, BCN, and their two-phase combination.

methods. For example, the ACC, F1, PRE, and REC of the BCN are 98.25%, 98.17%, 98.14%, and 98.25%, respectively. They are all $\sim 2\%$ higher than that of the 1DCNN+LSTM. We also depict the confusion matrices of the best four methods (i.e., the BCN, 2DCNN+LSTM, DCNN, and MCNN) in Fig. 8. These results also confirm the superiority of BCN as the confusion matrix of the BCN in Fig. 8a has the darkest main diagonals.

D. Two-Phase Detection Performance

Fig. 9 illustrates the validation performance on the controller and the H3C switch. On the controller, the DT is built with the scikit-learn, and the BCN is built with the Pytorch. On the H3C switch, the DT is implemented in P4, and the BCN is run with the MNN. Also, the validation traffic in the UNSW-NB15 dataset is replayed to go through the different devices.

The DT is only responsible for the binary-class detection, the BCN performs a multi-class detection, and the DT+BCN denotes the proposed two-phase detection. As shown in Fig. 9, the results on the controller and the switch have the same values, which confirms the correctness of our model implementation. For example, the ACC of the binary-class DT and the DT+BCN are 98.75% and 98.25%, which imply a high quality of the filtered malicious packets and the rationality of our two-phase detection.

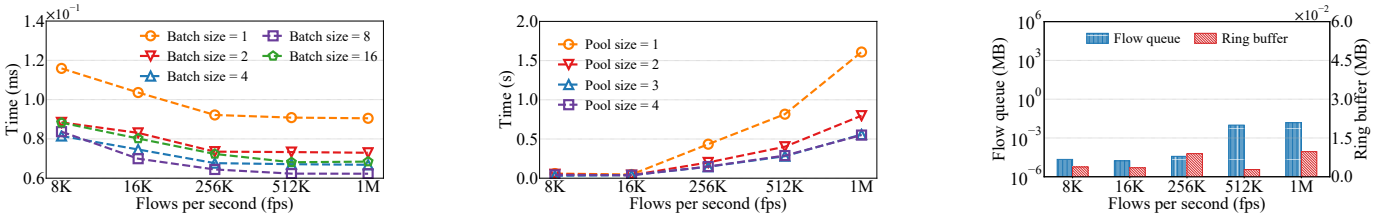
E. Prototype Analysis

Now we use the artificial traffic of $8K \sim 1M$ flows per second (fps) under the speed of 40Gbps and 100Gbps (10 scenarios in total) to evaluate the performance of Soter. By observation on the UNSW-NB15 dataset, the DT detects 5% of the validation traffic to be illegitimate. Therefore, we also clone 5% of the generated traffic to the intensive detection module for malicious simulation.

Aiming at running on low performance switches, the BCN reduces the floating-point complexities such as FLOPs and Para (in Fig. 6), achieving a fast speed. Fig. 10a and 11a both show that under the traffic of 1Mfps, detecting a batch of eight flows costs the lowest time, i.e., $0.06ms$ per flow, which is even more efficient than some GPU works (e.g., $2ms$ in [13]).

In Section VI-A, we instantiate several BCN processes in the BCN pool to accelerate the detection process. Here, we depict the average finish time for a flow from being cloned to the CPU to completing its detection in the BCN in Fig. 10b and 11b. The results show that the finish time decreases as the number of BCN models increases from one to three. For example, in Fig. 11b, under the flow rate of 16Kfps, the lowest time ($0.03s$) occurs when the pool size is three (i.e., three BCNs). However, the performance of four BCNs is equal to (in Fig. 10b) or even weaker (in Fig. 11b) than that of three BCNs. The main reason is that the process synchronization overhead outweighs the parallel acceleration as all the BCN models compete for access to the flow queue.

Fig. 10c and 11c illustrate the memory consumption of the flow queue and ring buffer in the intensive detection module. As the traffic is generated under the IMIX mode (the packet length varies from 64 to 1518), the memory consumption fluctuates in different experiments where the flow rate ranges from 16Kfps to 1Mfps. But overall, the consumption is positively correlated with the flow rate. Note that the memory consumption is very low in these simulation experiments, e.g.,

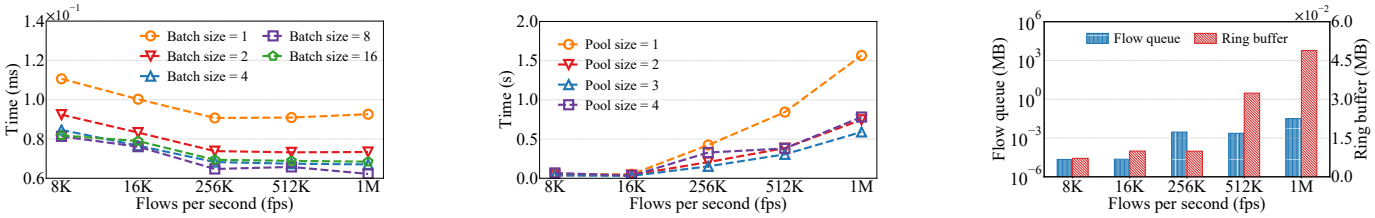


(a) The per-flow detection time of the BCN

(b) The per-flow finish time of the intensive detection module

(c) Memory per second

Fig. 10: The average performance under the traffic of 40Gbps.



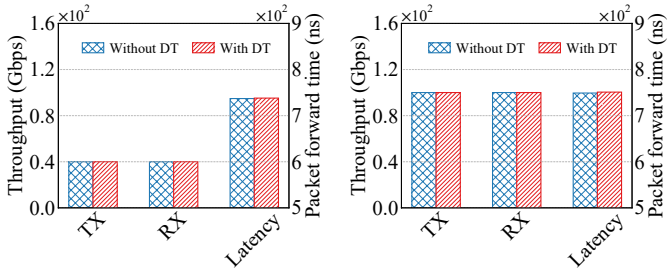
(a) The per-flow detection time of the BCN

(b) The per-flow finish time of the intensive detection module

(c) Memory per second

Fig. 11: The average performance under the traffic of 100Gbps.

the ring buffer consumes $\sim$ 0.05MB when the flow rate is 1Mfps in Fig. 11c. There are several reasons. First, thanks to the two-phase detection, only a small fraction (5% in the experiments) of the flows are cloned to the CPU. Second, we only store three packets for each flow, the extra packets of a flow are removed by the bloom filter. In addition, the multi-core processing improves the overall system efficiency, which reduces the buffering requirement.



(a) 40Gbps

(b) 100Gbps

Fig. 12: The switch throughput and packet forwarding time with or without the DT (pre-screening module).

Fig. 12 shows the receive (RX) throughput, transmit (TX) throughput, and packet forwarding time (latency) of the H3C switch (with or without our pre-screening module in Section VII). As shown, the pre-screening module adds little to the packet processing overhead of the Tofino ASIC while efficiently reducing the packet flooding to the CPU for the expensive DL detection. In addition, as packets are cloned but not redirected to the CPU, the intensive detection module does not affect the overall packet forwarding.

IX. Conclusion and Discussion

In this paper, we propose Soter, the DL enhanced in-network attack detection solution. Soter leverages a two-phase attack detection to jointly consider the detection accuracy and throughput. A DT-based P4 program runs on the Tofino ASIC to filter packets. Meanwhile, the intensive detection module runs on the switch CPU, utilizing the lightweight BCNs for the DL-based detection on suspicious packets. Experiments on a commodity switch illustrate that the BCN can identify a flow in $0.06ms$ with an accuracy of 98.25%, and the whole detection time for a flow is $0.03s$ with the traffic of 100Gbps and 16Kfps. According to the measurements [47]–[49] of different production data centers (e.g., university, enterprise, and cloud), a Top-of-Rack switch usually manages $10K \sim 100K$fps. Therefore, we argue that Soter has shown promising results for deep learning-based in-network applications.

Nevertheless, the CPU module still could be a bottleneck when facing a high load. If so, we suggest using the drop instead of the clone instruction in the DT (Listing 2) for the mitigation, sacrificing the accuracy but guaranteeing the real-time processing of the benign traffic.

X. Acknowledgment

This work is supported by the National Key Research and Development Project of China under grant No. 2020AAA0107704, National Natural Science Foundation of China under grant No. 61972189 and 62073263, Shenzhen Key Lab of Software Defined Networking under grant No. ZDSYS20140509172959989, and Research Center for Computer Network (Shenzhen) Ministry of Education.

REFERENCES

[1] R. SaiSindhuTheja and G. K. Shyam, "An efficient metaheuristic algorithm based feature selection and recurrent neural network for dos attack detection in cloud computing environment," *Applied Soft Computing*, vol. 100, p. 106997, 2021.

[2] S. Q. A. Shah, F. Z. Khan, and M. Ahmad, "The impact and mitigation of ICMP based economic denial of sustainability attack in cloud computing environment using software defined network," *Computer Networks*, vol. 187, p. 107825, 2021.

[3] S. M. Kerner, "Ddos attack snarls friday morning internet traffic," Website, https://www.eweek.com/security/ddos-attack-snarls-friday-morning-internet-traffic/, accessed: 2022-02-24.

[4] P. Illy, G. Kaddoum, C. M. Moreira, K. Kaur, and S. Garg, "Securing fog-to-things environment using intrusion detection system based on ensemble learning," in *Proceedings of the 2019 IEEE Wireless Communications and Networking Conference*. IEEE, 2019, pp. 1–7.

[5] D. Kwon, K. Natarajan, S. C. Suh, H. Kim, and J. Kim, "An empirical study on network anomaly detection using convolutional neural networks," in *Proceedings of the 38th IEEE International Conference on Distributed Computing Systems, 2018*. IEEE Computer Society, 2018, pp. 1595–1598.

[6] A. Meliboev, J. Alikhanov, and W. Kim, "1d CNN based network intrusion detection with normalization on imbalanced data," in *Proceedings of the 2020 International Conference on Artificial Intelligence in Information and Communication*. IEEE, 2020, pp. 218–224.

[7] A. Aleesa, M. Younis, A. A. Mohammed, and N. Sahar, "Deep-intrusion detection system with enhanced unsw-nb15 dataset based on deep learning techniques," *Journal of Engineering Science and Technology*, vol. 16, pp. 711–727, 2021.

[8] W. Wang, M. Zhu, X. Zeng, X. Ye, and Y. Sheng, "Malware traffic classification using convolutional neural network for representation learning," in *Proceedings of the 2017 International Conference on Information Networking*. IEEE, 2017, pp. 712–717.

[9] M. Belouch, S. El Hadaj, and M. Idhammad, "A two-stage classifier approach using reptree algorithm for network intrusion detection," *International Journal of Advanced Computer Science and Applications*, vol. 8, pp. 389–394, 2017.

[10] N. Moustafa and J. Slay, "The evaluation of network anomaly detection systems: Statistical analysis of the UNSW-NB15 data set and the comparison with the KDD99 data set," *Information Security Journal: A Global Perspective*, vol. 25, pp. 18–31, 2016.

[11] N. Ma, X. Zhang, H. Zheng, and J. Sun, "Shufflenet V2: practical guidelines for efficient CNN architecture design," in *Proceedings of the 15th European Conference on Computer Vision, 2018*. Springer, 2018, pp. 122–138.

[12] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the 2018 IEEE Conference on Computer Vision and Pattern Recognition*. Computer Vision Foundation / IEEE Computer Society, 2018, pp. 4510–4520.

[13] G. Xie, Q. Li, and Y. Jiang, "Self-attentive deep learning method for online traffic classification and its interpretability," *Computer Networks*, p. 108267, 2021.

[14] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[15] T. Pan, N. Yu, C. Jia, J. Pi, L. Xu, Y. Qiao, Z. Li, K. Liu, J. Lu, J. Lu, E. Song, J. Zhang, T. Huang, and S. Zhu, "Sailfish: accelerating cloud-scale multi-tenant multi-service gateways with programmable switches," in *Proceedings of the 2021 ACM SIGCOMM Conference*. ACM, 2021, pp. 194–206.

[16] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. M. Caulfield, E. S. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. G. Greenberg, "Azure accelerated networking: Smartnics in the public cloud," in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation, 2018*. USENIX Association, 2018, pp. 51–66.

[17] Z. Xiong and N. Zilberman, "Do switches dream of machine learning?: Toward in-network classification," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*. ACM, 2019, pp. 25–33.

[18] J.-H. Lee and K. Singh, "Switchtree: in-network computing and traffic analyses with random forests," *Neural Computing and Applications*, pp. 1–12, 2020.

[19] C. Zheng and N. Zilberman, "Planter: seeding trees within switches," in *Proceedings of the 2021 ACM SIGCOMM Conference*. ACM, 2021, pp. 12–14.

[20] C. Busse-Grawitz, R. Meier, A. Dietmüller, T. Bühler, and L. Vanbever, "pforest: In-network inference with random forests," *CoRR*, vol. abs/1909.05680, 2019.

[21] B. M. Xavier, R. S. Guimar aes, G. Comarela, and M. Martinello, "Programmable switches for in-networking classification," in *Proceedings of the 40th IEEE Conference on Computer Communications, 2021*. IEEE, 2021, pp. 1–10.

[22] 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 *Proceedings of the 41th IEEE Conference on Computer Communications, 2022*. IEEE, 2022, pp. 1938–1947.

[23] D. Barradas, N. Santos, L. Rodrigues, S. Signorello, F. M. V. Ramos, and A. Madeira, "Flowlens: Enabling efficient flow classification for ml-based network security applications," in *Proceedings of the 28th Annual Network and Distributed System Security Symposium*. The Internet Society, 2021.

[24] P. Cui, H. Pan, Z. Li, J. Wu, S. Zhang, X. Yang, H. Guan, and G. Xie, "Netfc: Enabling accurate floating-point arithmetic on programmable switches," in *Proceedings of the 29th IEEE International Conference on Network Protocols, 2021*. IEEE, 2021, pp. 1–11.

[25] W. Wang, M. Zhu, J. Wang, X. Zeng, and Z. Yang, "End-to-end encrypted traffic classification with one-dimensional convolution neural networks," in *Proceedings of the 2017 International Conference on Intelligence and Security Informatics*. IEEE, 2017, pp. 43–48.

[26] M. Lopez-Martin, B. Carro, A. Sanchez-Esguevillas, and J. Lloret, "Network traffic classifier with convolutional and recurrent neural networks for internet of things," *IEEE Access*, vol. 5, pp. 18 042–18 050, 2017.

[27] N. Moustafa and J. Slay, "UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)," in *Proceedings of the 2015 Military Communications and Information Systems Conference*. IEEE, 2015, pp. 1–6.

[28] R. K. Malaiya, D. Kwon, S. C. Suh, H. Kim, I. Kim, and J. Kim, "An empirical evaluation of deep learning for network anomaly detection," *IEEE Access*, vol. 7, pp. 140 806–140 817, 2019.

[29] Y. Yang, K. Zheng, C. Wu, and Y. Yang, "Improving the classification effectiveness of intrusion detection by using improved conditional variational autoencoder and deep neural network," *Sensors*, vol. 19, p. 2528, 2019.

[30] S. Naseer, Y. Saleem, S. Khalid, M. K. Bashir, J. Han, M. M. Iqbal, and K. Han, "Enhanced network anomaly detection based on deep neural networks," *IEEE Access*, vol. 6, pp. 48 231–48 246, 2018.

[31] Y. Yu, J. Long, and Z. Cai, "Network intrusion detection through stacking dilated convolutional autoencoders," *Security and Communication Networks*, vol. 2017, pp. 4 184 196:1–4 184 196:10, 2017.

[32] I. Corporation, "Tofino switch," Website, https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html, accessed: 2022-03-02.

[33] A. Z. Broder and M. Mitzenmacher, "Survey: Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, pp. 485–509, 2003.

[34] G. Chandrashekar and F. Sahin, "A survey on feature selection methods," *Computers and Electrical Engineering*, vol. 40, pp. 16–28, 2014.

[35] S. Khalid, T. Khalil, and S. Nasreen, "A survey of feature selection and feature extraction techniques in machine learning," in *Proceedings of the 2014 Science and Information Conference*. IEEE, 2014, pp. 372–378.

[36] M. Lotfollahi, M. J. Siavoshani, R. S. H. Zade, and M. Saberian, "Deep packet: a novel approach for encrypted traffic classification using deep learning," *Soft Computing*, vol. 24, no. 3, pp. 1999–2012, 2020.

[37] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapè, "Toward effective mobile encrypted traffic classification through deep learning," *Neurocomputing*, vol. 409, pp. 306–315, 2020.

[38] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proceedings of the 1st International Conference on Learning Representations, 2013*, 2013.

[39] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning, 2015*. JMLR.org, 2015, pp. 448–456.

[40] A. Howard, R. Pang, H. Adam, Q. V. Le, M. Sandler, B. Chen, W. Wang, L. Chen, M. Tan, G. Chu, V. Vasudevan, and Y. Zhu, "Searching for mobilenetv3," in *Proceedings of the 2019 IEEE/CVF International Conference on Computer Vision*. IEEE, 2019, pp. 1314–1324.

[41] D. Scherer, A. C. Müller, and S. Behnke, "Evaluation of pooling operations in convolutional architectures for object recognition," in *Proceedings of the 20th International Conference on Artificial Neural Networks, 2010*. Springer, 2010, pp. 92–101.

[42] P. L. Consortium, "Behavioral model (bmv2)," Website, https://github.com/p4lang/behavioral-model, accessed: 2022-03-01.

[43] R. Bolla, R. Bruschi, C. Lombardo, and D. Suino, "Evaluating the energy-awareness of future internet devices," in *Proceedings of the 12th IEEE International Conference on High Performance Switching and Routing, 2011*. IEEE, 2011, pp. 36–43.

[44] S. Communications, "Imix (internet mix)," Website, http://www.spirent.com/~/media/TestMethodologyJournal/Spirent_TestMethodology_IMIX-Journal.pdf, accessed: 2022-03-01.

[45] X. Jiang, H. Wang, Y. Chen, Z. Wu, L. Wang, B. Zou, Y. Yang, Z. Cui, Y. Cai, T. Yu, C. Lyu, and Z. Wu, "MNN: A universal and efficient inference engine," in *Proceedings of Machine Learning and Systems*. mlsys.org, 2020, pp. 1–13.

[46] A. Appleby, "Murmurhash," Website, https://sites.google.com/site/murmurhash/, accessed: 2022-02-24.

[47] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM Internet Measurement Conference, 2010*. ACM, 2010, pp. 267–280.

[48] S. Kandula, S. Sengupta, A. G. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proceedings of the 9th ACM SIGCOMM Internet Measurement Conference, 2009*. ACM, 2009, pp. 202–208.

[49] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: fine grained traffic engineering for data centers," in *Proceedings of the 2011 Conference on Emerging Networking Experiments and Technologies*. ACM, 2011, p. 8.