Computer Communications 182 (2022) 153–169

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

On-the-fly (D)DoS attack mitigation in SDN using Deep Neural Network-based rate limiting

Ali El Kamel [*], Hamdi Eltaief, Habib Youssef

PRINCE Research Lab. ISITC Hammam Sousse, University of Sousse, Tunisia

ARTICLE INFO

Keywords:
Software Defined Networking (SDN)
Deep/Shallow Neural Networks (SNN)
MultiLoss SNN (ML-SNN)
Online Deep Learning (ODL)
Rate limiting
Trustworthiness

ABSTRACT

Software Defined Networking (SDN) has emerged as a promising paradigm offering an unprecedented programmability, scalability and fine-grained control over forwarding elements (FE). Mainly, SDN decouples the forwarding plane from the control plane which is moved to a central controller that is in charge of taking routing decisions in the network. However, SDN is rife with vulnerabilities so that several network attacks, especially Distributed Denial of Service (DDoS), can be launched from compromised hosts connected to switches. DDoS attacks can easily overload the controller processing capacity and flood switch flow-tables.

This paper deals with the security issue in SDN. It proposes a real-time protection against DDoS attacks that is based on a controller-side sliding window rate limiting approach which relies on a weighted abstraction of the underlying network. A weight defines the allowable amount of data that can be transmitted by a node and is dynamically updated according to its contribution to: (1) the queueing capacity of the controller, and (2) the number of flow-rules in the switch. Hence, a new deep learning algorithm, denoted the Parallel Online Deep Learning algorithm (PODL), is defined in order to update weights on the-fly according to both aforementioned constraints simultaneously. Furthermore, the behavior of each host and each switch is evaluated through a measure of trustworthiness which is used to penalize mis-behaving ones by prohibiting new flow requests or PacketIn messages for a period of time. Host trustworthiness is based on their weights while switch trustworthiness is achieved through a computation of the Average Nearest-Neighbor Degree (ANND). Realistic experiments show that the proposed solution succeeds in minimizing the impact of DDoS attacks on both the controllers and the switches regarding the PacketIn arrival rate at the controller, the rate of accepted requests and the flow-table usage.

1. Introduction

Software Defined Networking (SDN) [1,2] is considered as a promising technology towards network evolution and programmability by decoupling data and control planes. Basically, SDN provides network resources with sharing, flexibility, adaptability and fine-grained control over switches at less cost compared to conventional IP networks.

SDN consists of a central controller with a global visibility of the network state that can order updates to any switch directly once required. The communication between the controller and the switch is handled by open and standard protocols such as OpenFlow [3].

In OpenFlow-based networks, the controller-to-switch communication takes place as follows: As the first packet of a flow reaches the ingress switch, a ***PacketIn*** message is sent to the controller if no matching rule is found in the local flow-table. Upon receiving a ***PacketIn*** message, the controller analyzes the request, computes the forwarding path and updates all nodes within it by adding entries to the corresponding flow tables. All subsequent packets of the flow are forwarded based on new added entries without any need to contact the controller. Updates are carried into ***PacketOut*** or ***FlowMod*** messages.

Unfortunately, SDN is rife with vulnerabilities [4,5] so that several network attacks, especially Distributed Denial of Service (DDoS), may be launched from malicious hosts connected to switches. DDoS attacks can easily target an SDN network by overloading the queue of the controller or flooding the flow-table of the switch. Typically, overloading a controller consists of sending an excessive amount of ***PacketIn*** messages from multiple switches so that the controller can no longer accept new incoming requests leading to service breakdown.

When a controller receives a new PacketIn message, it establishes a path joining the source to the destination. Then, it installs new flow-entries in flow-tables of all crossed switches in order to keep packet forwarding. Nevertheless, flow-tables are proclaimed to be expensive and with limited capacity [6]. This limitation opens a door for DoS attacks on switches in which the flow-table is flooded using a large number of new flow-entries that exceeds the available memory space. Today, mitigating DDoS attacks in SDN is with paramount importance.

* Corresponding author.
E-mail address: ali.kamel@isitc.u-sousse.tn (A. El Kamel).

https://doi.org/10.1016/j.comcom.2021.11.003
Received 30 May 2020; Received in revised form 26 September 2021; Accepted 4 November 2021
Available online 17 November 2021

The DDoS mitigation issue is one of the hardest problems that network operators face, especially in SDN. DDoS attack mitigation aims at ensuring that the controller is not overwhelmed even when it receives a huge amount of PacketIn messages. This can be achieved through common techniques of rate limiting e.g token bucket, leaky bucket, fixed window and sliding window. Without rate limiting, each host may send new flow requests as often as it likes, which can lead to a substantial amount of requests that starve the controller. After enabling rate limiting, each host is limited to a fixed number of requests for each period of time. However, rate limiting does not distinguish between benign and malicious hosts and has no mechanisms to determine whether a connection is legitimate or not. Moreover, security policies of a rate limiter are applied to all hosts indifferently so that benign hosts are penalized likewise malicious ones, leading to a high degree of false positives. Furthermore, common techniques of rate limiting does not actually block any bad traffic initiated by a malicious adversary. Bad traffic will reach the controller, albeit at a slower rate. Finally, all rate limiting mechanisms are based on the specification of limits a priori which may penalize bandwidth-sensitive applications. To deal with above drawbacks, a rate limiting mechanism must be dynamically defined on a per-host basis and must depend on the time-series forecasting of both the controller queueing capacity and the switch memory free space. This can be achieved through Online Deep Learning (ODL).

Online Deep Learning (ODL) [7] is a subsection of machine learning in which data becomes available in a sequential order and is used to update the best prediction model for future data at each step, as opposed to batch learning techniques which generate the best prediction model by learning on the entire training data set at once. ODL is a method of updating Deep Neural Networks (DNN) on the fly.

This paper deals with the DDoS attack mitigation issue in SDN. It is based on the combination of a rate-limiting logic with deep learning capabilities so that DDoS attacks can be avoided promptly. The rate-limiting logic consists of restricting the amount of incoming PacketIn messages to the controller. The number of new flow requests of all hosts are input to a Shallow Neural Network (SNN) which predicts the expected number of received PacketIn messages in the controller through the definition of weights on links between devices. A weight defines the amount of new flow requests sent by a host according to both the available queue capacity of the controller and the available memory space in the switch. Weights are adjusted through an Online Deep Learning (ODL) algorithm. If the weight associated to a host goes under a fixed threshold, the host is considered as a malicious entity and it is put in blocked mode.

In the same way, a switch is considered as compromised by computing its trustworthiness so that it is forbidden to send new PacketIn messages for a period of time. Trustworthiness is a measure of cohesiveness and correlation of a switch among its neighborhood and is evaluated through a weighted version of the correlation degree denoted the Average Nearest-Neighbor Degree (ANND) [8]. More details are given in Section 5.

The rest of the paper is structured as follows: Section 2 describes limits of existing approaches dealing with the DDoS mitigation problem. Section 3 describes the background of our approach and proposes an analytic formulation of the problem. Section 4 introduces the proposed approach and details technical aspects. Section 5 presents the paradigm of DDoS attacks mitigation using trustworthiness assessment. Performance evaluation of the proposed algorithms are stated in Section 6. Section 7 presents simulation results of the whole approach. Section 8 states concluding remarks and perspectives.

2. Related work

With the increasing adoption of SDN technology, considerable efforts have being devoted to address security issues and to offer mitigation solutions. Authors in [9] have provided a survey on these issues. Particularly, researches in [10–15] have focused mainly on research efforts for mitigating (D)DoS attacks in SDN networks.

2.1. Classical DDoS mitigation in SDN

Authors in [10] and in [11] have evaluated the impact of DoS attacks on the network performance parameters like the control plane bandwidth (i.e., controller-switch channel), latency, switches flow tables and the controller performance, without providing a solution to overcome these issues. In [12], authors have proposed FlowRanger, a controller-side scheme that allows to detect and mitigate DoS attacks. It consists of: (1) calculating a trust value for each packet-in message based on its source, (2) queueing the message in the priority queue corresponding to its trust value and (3) processing messages according to a weighted Round Robin strategy. Although FlowRanger can reduce the impact of DoS attacks on network performance by prioritizing legitimate flows for processing in spite of malicious one, it does neither prevent flooding the controller nor limit overloading switch's flow-tables.

Sahay et al. [13] have proposed a self-management scheme which is based on a cooperation between an ISP and its customers in order to mitigate DoS attacks. Mainly, the ISP collects costumer's experiences regarding network performance parameters such as controller response time and exploits them to enforce security policies and update flow tables in the network accordingly. Therefore, legitimate flows are prioritized and are routed through a path with higher quality. Inversely, if there is a doubt about the legitimacy of the flow, the ISP controller will assign a low priority to the flow and direct through the path designated for malicious flows. Unfortunately, this scheme cannot mitigate the risks of flooding the controller and overloading flow-tables in the switches. Dao et al. [14] have used IP filtering to protect SDN networks against distributed DoS attacks. It consists of assigning short timeouts for flow-entries which IP source (user) is considered a malicious source and long timeouts are used for trusted ones. Although this solution can drop all malicious traffic, it may drop legitimate traffic if the flow duration is higher than the assigned timeout.

Finally, Authors in [15] have presented the SDN-Guard solution. It consists of protecting SDN networks against DoS attacks by dynamically (1) rerouting potential malicious traffic, (2) adjusting flow timeouts and (3) aggregating flow rules. The solution uses an IDS to measure the threat probability of each new incoming flow. However, this scheme introduces communication overhead to the control plane due to the exchange of messages between the controller, the IDS and network functions deployed within the cloud.

2.2. ML-based DDoS mitigation in SDN

Authors in [16] propose a SDN-based framework to detect and defense DDoS attacks using machine learning. The framework extracts traffic characteristics through statistically analyzing flow-tables of the switches. Then, characteristics are inputted to SVM algorithm to classify the flow. Results are then used to adjust forwarding policies. In the same way, authors in [17] suggest using SVM to identify DDoS attacks by extracting features from PacketIn messages. Then, the entropy of each feature is measured. In case of high entropy, the trained SVM algorithm is used to classify the flow into a legitimate flow or an attack.

2.3. DL-based DDoS mitigation in SDN

In [18], a systematic benchmarking analysis of the existing machine learning techniques for the detection of malicious traffic in SDNs is proposed. Authors identify the limitations in these classical machine learning based methods, and suggest several assumptions for the foundation for a more robust framework. Because of irrelevant features and the lack of labeled training data, results show a poor classification and difficulties in detecting the attacks when classical ML-techniques are used. They reveal that deep learning (DL) is one of the most promising solutions for solving the weaknesses of machine learning.

Deep learning algorithms have been widely used in SDN-based architectures to solve the problem of DDoS mitigation and intrusion detection. In [19], authors present a Gated Recurrent Unit Recurrent Neural Network (GRU-RNN) which enables intrusion detection systems for SDNs. Authors in [20] propose a novel deep learning based Convolutional Neural Network (CNN) approach for feature detection. In [21], authors apply a deep learning approach for flow-based anomaly detection in an SDN environment.

2.4. RL-based DDoS mitigation in SDN

Several researches view the mitigation of DDoS attacks as a sequential decision making problem and use Deep Reinforcement Learning techniques to achieve the aim. In [22], authors propose "Multiagent Router Throttling". It consists of installing multiple reinforcement learning agents on a set of routers that learn to throttle or rate-limit traffic towards a victim server. Yandong et al. [23] propose a smart deep reinforcement learning based framework, which can mitigate the DDoS flooding attacks in real time. The proposed framework can defend against a wide range of DDoS flooding attacks such us TCP SYN, UDP, and ICMP flooding. Mainly, it learns the patterns of all the traffic in order to throttle the attack traffic and keep the traffic of benign users unaltered. Authors in [24] adopt reinforcement learning to the L7 DDoS problem, which is viewed as a Markov decision process. The proposed approach inherits advantages from both learning-based approaches and Markov models and incorporates environmental and contextual factors to distinguish L7 DDoS traffic from the legitimate application-layer traffic.

However, authors in [23] revealed that RL-based methods cannot be extended to a real large-scale SDN network because discretizing the continuous action space leads to the combinatorial explosion and the well-known curse of dimensionality. Despite that RL is a promising technique to mitigate DDoS attacks, little efforts were devoted to define RL-based approaches to countermeasure cyber-attacks. Most of solutions employ deep learning to learn from historical data and identify the attack traffic (at least until the writing of those lines).

2.5. OpenFlow-Based DDoS mitigation in SDN

Authors in [25] assume that the main cause of DDoS attacks in SDN is compromised hosts. They reveal also that compromised hosts exploit the vulnerabilities of Openflow to propagate their attacks. To deal with this issue, they propose a time and space-efficient solution for the identification of these compromised hosts. They prove through simulation that the solution consumes less computational resources and space and does not require any special equipment.

Many studies were done to detect DDoS attacks on SDN due to vulnerabilities in Openflow [26,27]. Authors in [28] propose a secure controller-to-controller (C-to-C) protocol that allows SDN-controllers lying in different autonomous systems (AS) to securely communicate and transfer attack information with each other. Kotani et al. [29] present a solution to filter out Packet-In messages to be processed by the centralized controller to save it from getting overburdened. Their solution suggests applying restrictions on important and less important messages based on the value in headers. The switches store the header values of the packets to be forwarded to the controller for time t. If the packet with existing header values arrives in t, it does not get forwarded to the controller in that time window. This solution does not cater for DDoS attack with non-repetitive random header patterns.

Mousavi et al. [30] introduce the idea of using entropy to identify the decrease in randomness in the flow of packets towards controllers and detect an early DDoS attack. The frequencies for the destination IP addresses are stored for window size t and entropy is calculated to identify the DDoS attack when the arrived packet number reaches the window size.

2.6. Using load-balancing techniques in mitigating control-plane DDoS

From another hand, several researches focus on improving dynamic switch-to-controller assignment and think that this may mitigate DoS attacks on controllers. Indeed, they conclude that achieving load balancing among controllers helps to reduce the risk of control-plane DoS attacks since they can anticipate controller overloading and so migrate switches from overloaded controllers to under-loaded ones.

Authors in. [31] propose a dynamic controller provisioning as an Integer Linear Problem in order to minimize the flow setup time and the communication overhead. Mainly, this work is based on reassigning switches to controllers by optimizing three constraints related to switch-to-controller exchange cost, inter-controller synchronization cost and switch reassignment cost, respectively. Authors in [32] focus on network partitioning as a solution for the controller placement issue. Based on two constraints: the switch-to-controller latency and the path survivability, a multi-objective optimization problem is there defined.

In [33], a new architecture of distributed controllers is proposed. Based on current traffic conditions, the pool of controllers is updated. Moreover, this paper proposes a new approach to migrate switches from an overloaded controller to another less loaded. Authors in [34] solve the switch-to-controller assignment issue using game theory for powerful decision making. It consists of a many-to-one matching phase followed by a coalition game to achieve load balancing among controllers. However, achieving load balancing is considered very costly in terms of network load since switch migration between controllers requires exchanging several messages.

A new approach is presented in [35] which associates switches to controllers while maintaining balanced loads among controllers. This approach aims to minimize the whole setup time of incoming flows by achieving a trade-off between the switch-to-controller round trip time (RTT) and the current loads on a controller. The migration is achieved with no communication overload since no updating messages are exchanged. Mainly, the problem is formulated as a minimum cost bipartite assignment optimization problem and solved using an improvement of the Hungarian Algorithm. This work is able to maintain a load balancing state between controllers leading to better lifetime. Clearly, load-balancing approaches may help to protect controllers against DDoS threats by balancing their loads, however, non of them have focused on switch-side DDoS attacks.

2.7. Critics

Although the aforementioned solutions were able to protect the control plane against DDoS attacks, they do not provide a solution to switch's TCAM exhaustion, except SDN-GUARD [15]. Indeed, most of the solutions introduce complexity to the network by adding new components (e.g [12,13,15]), and/or by exchanging control messages (e.g [15,28]) which may lead to a significant communication overload. Our proposed solution aims to mitigate DDoS attacks on controllers and switches without installing new extra components in the network and without introducing communication overload, using a deep learning-based anticipation of misbehaving nodes and a rate-limiting scheme based on a penalization factor. More details are given in Section 4. Table 1 summarizes related work.

3. Background and problem formulation

In this section, a brief description of SDN paradigm is presented. Then, the concept of DDoS attacks and their impacts on the network performance are analyzed. Thus, drawbacks and limitations of common techniques of rate limiting are described. Finally, the concept of Online Deep Learning is introduced before presenting a formulation of the problem.

Table 1
Recap of existing solutions.

Approach	Limits	Controller protection	Switch protection
Wei et al. [12]	- It does not support QoS due to the use of RR scheduling, - It cannot mitigate IP spoofing-based DDoS attacks	Yes	No
Sahay et al. [13]	- The costumer experience can be falsified - The frequency of collecting costumers experiences is static - It can lead to unbalanced network	Yes	No
Dao et al. [14]	- It achieves low accuracy due to a high rate of false positive - Cannot mitigate IP spoofing-based DDoS attacks	No	No
Dridi et Zhani. [15]	- it introduces communication overload to the control plane - require deploying an IDS within the cloud.	Yes	Yes
Hameed and khan. [28]	- Require exchanging attack information between controllers - security of the protocol is not proved	No	No
Kotani et al. [29]	- This solution does not cater for DDoS attack with non-repetitive random header patterns	No	No
Mousavi et al. [30]	- Entropy based DDoS attack mitigation using Ip destination addresses - a static Fixed window for counting the arrival packet number.	Yes	No

3.1. SDN paradigm

SDN [1] is recognized as an emerging technology towards network communication softwarization. In SDN, the network becomes dynamic, programmable, flexible and self-adaptive according to environmental requirements [2,36,37]. Mainly, SDN is based on decoupling the control plane from the data/forwarding plane so that computing capabilities are moved to a centralized controller. The controller has a global view of the network state that helps it to make correct decisions considering the whole situation.

The SDN architecture is described in Fig. 1 [38]. In conventional IP networks, the control plane and the data plane are hosted within the same device. Hence, forwarding data is managed locally without having a global view on the network state. As a result, resources are not optimally utilized leading to under-utilized links and overwhelmed ones. In SDN, the control planes of all devices are combined and hosted within the same device, called the controller, which takes decisions on data forwarding and transmits flow rules to switches through standard and open protocols such as Openflow [3].

SDN consists of three layers: an application layer, a control layer and a forwarding layer. The application layer includes applications of management and security of the underlying network. The control layer consists of centralized controllers that connect the application layer to the underlay forwarding layer. Known as the brain of network, the controller performs processing locally or using the application layer and instructs decisions to switches for their operation. Finally, the forwarding layer consists of physical network entities (switches) that are controlled and managed according to the application specifications and requirements via the SDN controller.

3.2. DDoS Attacks on SDN controllers

Unfortunately, the centralization of control capabilities raises many security issues. Clearly, if the SDN controller is compromised, it can no more reply to the switches queries, so that, the whole network will be unavailable. An SDN controller may attract DDoS attacks [39], where a large number of new-flows with distinctive headers are injected from a group of malicious adversaries through a compromised (zombies) hosts [40,41]. Since it is the first-time that those flows are introduced in the switch, no flow matches are found in flow tables and information is forwarded to the controller for processing. As the number of new-flows increases, the processing load on the controller goes up leading to processing overload since the controller has a limited processing capacity. As a result, the controller becomes unavailable for the normal queries processing. A DDoS attack results in a degradation of the performance of the network [42,43].

3.3. Openflow protocol and its vulnerabilities

OpenFlow is a southbound protocol that maintains all the communication between the switches and the controller. Although an Openflow-based network offers a lot of benefits compared to the traditional network, it is rife with vulnerabilities [44] and it still needs improvement to make it secure. Basically, the original specification of Openflow protocol advises using Transport Layer Security (TLS) to protect the switch-to-controller communication. However, the subsequent specifications, including the last ones, consider the adoption of TLS for Openflow as an optional feature and not a mandatory requirement. The decision of skipping TLS is due to the complex procedure required to establish a TLS channel, which starts by generating both certificates of switches and controllers, signing them with the site-wide private key, and finally installing them into all of the devices. Comparatively, the basic Openflow operation only requires the controller address to connect the switch.

The lack of TLS leaves an open door for adversaries to attack Openflow-based networks [45,46]. An attacker places a compromised device between a controller and a switch and could intercept communication, insert additional rules into flow tables, gain access to protected segments of the network and immediately reconfigure all of the down-stream switches (Man-in-the-Middle attack). This is feasible since the connectivity between the switch and the controller is never interrupted unless one of them starts it. With the lack of authentication in basic Openflow, an attacker can immediately hijack full control of any downstream switches and execute very fine-grained malicious operations.

3.4. DDoS attacks on flow-tables

Each new-flow request is associated with a rule which is installed in all crossed switches from the source to the destination to keep subsequent packets forwarding. When a huge number of rules is installed in a flow-table, the switch is overwhelmed and it can no longer serve new flows. The switch is considered out-of-service for some time which, probably, may lead to service unavailability.

3.5. Rate limiting logic

Rate limiting is generally put in place as a defensive measure for services. Shared services need to protect themselves from excessive intended or unintended use to maintain service availability. For the system to perform well, clients must also be designed with rate limiting in mind to reduce the chances of cascading failure. At the network layer, Rate limiting is a frequently used solution to defend against

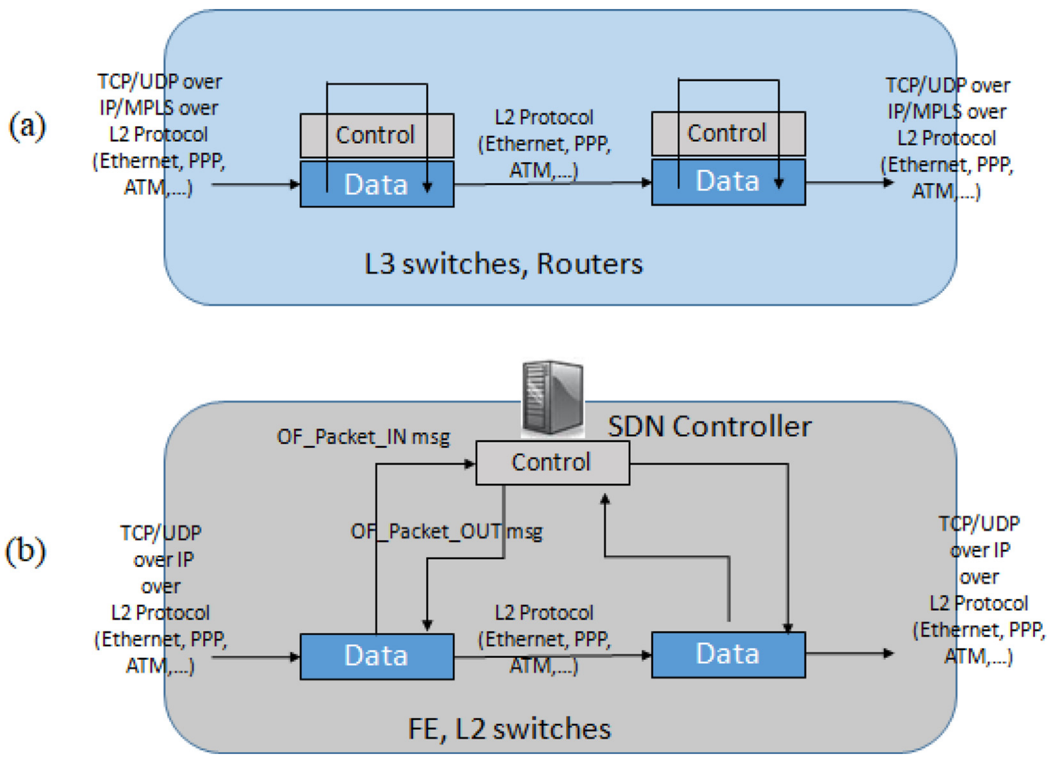
(a)
TCP/UDP over IP/MPLS over L2 Protocol (Ethernet, PPP, ATM,...)
Control
Data
L2 Protocol (Ethernet, PPP, ATM,...)
Control
Data
TCP/UDP over IP/MPLS over L2 Protocol (Ethernet, PPP, ATM,...)
L3 switches, Routers
(b)
SDN Controller
OF_Packet_IN msg
Control
OF_Packet_OUT msg
TCP/UDP over IP over L2 Protocol (Ethernet, PPP, ATM,...)
Data
L2 Protocol (Ethernet, PPP, ATM,...)
Data
TCP/UDP over IP over L2 Protocol (Ethernet, PPP, ATM,...)
FE, L2 switches

Fig. 1. (a) Conventional IP networks (b) Software Defined Networks (SDN).

network DDoS attacks. When the rate of incoming requests becomes too large for a sink to handle, rate limiting is applied to restrict the amount of incoming packets. Rate Limiting can often protect against brute-force password attempts, and other types of abusive behavior targeting either the network layer or the application layer [47].

There are several different techniques for measuring and limiting rates, each with their own uses and implications:

– **Token bucket**: A token bucket maintains a rolling and accumulating budget of usage as a balance of tokens. When a service request is made, the service attempts to decrement the token count to fulfill the request. If there are no tokens in the bucket, the service has reached its limit and the request is not satisfied.
– **Leaky bucket**: A leaky bucket is similar to a token bucket, but the rate is limited by the amount that can drip or leak out of the bucket. This technique recognizes that the system has some degree of finite capacity to hold a request until the service can act on it; any extra simply spills over the edge and is discarded.
– **Fixed window**: Fixed-window limits a number of requests per each period of time, but they are subject to spikes at the edges of the window, which might overwhelm the service.
– **Sliding window**: Sliding windows have the benefits of a fixed window, but the rolling window of time smooths out bursts. Usually, sliding-window is recognized as weight-based rate limiting.

Common techniques of rate limiting does not actually block any bad traffic initiated by a malicious adversary. Bad traffic will reach the controller, albeit at a slower rate. Mainly, all rate limiting mechanisms are based on the specification of limits a priori and on the application of a default action which consists of dropping all extra data when limits are reached. This may penalize both legitimate applications such as bandwidth-sensitive applications and benign hosts.

3.6. Deep neural network (DNN) and online deep learning (ODL)

Deep neural networks [48] are a powerful category of machine learning algorithms implemented by stacking layers of neural networks along the depth and width of smaller architectures. Deep networks have recently demonstrated discriminative and representative learning capabilities over a wide range of applications in the recent years.

Researchers in ML are expanding the horizons of deep learning by seeking their prospective applications in other diverse domains. Deep networks require a large amount of annotated data for training. With efficient training algorithms, deep neural networks are capable of separating millions of labeled images [49]. Moreover, the trained network can also be used for learning efficient image representations for other similar beneath data sets. When a training phase is not possible since the system cannot be interrupted, it is possible to go though online learning using Online Deep Learning algorithms (ODL).

Unlike batch learning, online learning represents a class of learning algorithms that learn to optimize predictive models over a stream of data instances in a sequential manner. The nature of on-the-fly learning makes online learning highly scalable and memory efficient [50].

3.7. Problem formulation

An SDN network consists of an undirected graph G = (C, S, H) where C = $(c_1, c_2, \dots, c_M)$ is the set of controllers, S = $(s_1, s_2, \dots, s_N)$ is the set of switches and $H = (h_1, \dots, h_P)$ defines the set of hosts. Each switch s_j is connected to one controller c_k through Openflow protocol. Table 2 presents a list of notations adopted in this paper.

Let D_{s_j} be the set of hosts connected to the switch s_j. Let r_{h_i}, r_{max} be the number of new flows transmitted from the host $h_i \in D_{s_j}$ to the switch s_j during a period of time t and the maximum allowable number of new flows that the host h_i can send to the switch s_j, respectively. If we suppose that all new flows have no matches in the ingress switch, the total number of PacketIn messages r_{s_j} which must be forwarded from the switch s_j to the controller for processing is as follows (Eq. (1)):

$$r_{s_j} = \sum_{h_i \in D_{s_j}} r_{h_i} \tag{1}$$

$s.t$

$$r_{h_i} \leq r_{max} \tag{2}$$

Table 2
Notations.

Notations	Meaning
$H(s_i)$	A set of all hosts that are served by the switch s_i
$S(c_k)$	A set of switches that are connected to the controller c_k
r_{s_i}	The number of requests forwarded by the switch s_i
r_{h_i}	The number of requests sent by the host h_i
r_{max}	The maximum allowed number of requests to be sent by hosts
U_{c_k}	The number of requests that can be admitted at the controller c_k
U_{s_k}	The number of requests that can be admitted at the switch s_k
w^h_{ij}	The weight of the link between a host h_i and switch sj
w^s_{ij}	The weight of the link between a switch s_i and a controller c_j
Γ^w_i	The average nearest-neighbor degree of a switch s_i regarding a weight set W

The controller c_k receives an amount of PacketIn messages from multiple switches. The number of PacketIn messages that a switch s_j can send to the controller must not exceed the available space in the flow-table U_{s_j}. Let $S_{c_k} \subseteq S$ be the subset of switches connected to the controller c_k. For a period of time t, the controller c_k receives a total amount of requests r_{c_k} as follows (Eq. (3)):

$$r_{c_k} = \sum_{s_j \in S_{c_k}} r_{s_j} \tag{3}$$

$s.t$

$$r_{s_j} \leq U_{s_j} \tag{4}$$

Let U_{c_k} be the available queue capacity of the controller c_k. Therefore, the total amount of received requests must not exceed the available queue capacity in the controller, otherwise, the controller is overwhelmed and a subset of requests will be rejected. The problem in Eq. (3) is considered as an optimization problem which consists of minimizing a loss function between the total number of received PacketIn messages and the available queue capacity of the controller. This can be expressed as follows (Eq. (5)):

$$\min_{r_{s_i}} Q_{c_k} \tag{5}$$

$s.t$

$$r_{s_i} \leq U_{s_i} \forall s_i \in S(c_k)$$

where $Q_{c_k} = r_{c_k} - U_{c_k}$ defines a loss function between the expected number of received PacketIn messages and the available queue capacity $U_{c_k}(t)$ of the controller c_k.

4. Proposed approach

4.1. Approach overview

The problem presented in (5) consists of minimizing the loss function Q by dynamically adjusting the allowable number of new flow requests in each host. This is achieved through a sliding window based rate limiting mechanism that aims at capping the allowable number of new flow requests that each host can send during a period of time. Capping is achieved through the definition of weights on all links joining hosts to switches. A weight is used to specify the allowable number of new flow requests that a host can send to its edge switch during a cycle. At the end of each cycle, the weight is adjusted according to the contribution of the number of new flow requests transmitted by the host on the controller queue capacity and the switch flow-table space observed during the last cycle.

Thereby, weights are adjusted using two discounts. The first discount is denoted ψ and is computed through a training model which predicts the number of received PacketIn messages at a controller then compares it with the available queue capacity using an MSE-based loss function. The second discount is denoted ϕ and is obtained through the evaluation of the disparity between the current available memory of the flow-table of a switch and the number of PacketIn messages to the controller for processing. The disparity is computed using a softmax-based cross entropy (CE) loss function. CE measures the distance between a distribution of the probabilities of the flow-table usage and the distribution of the thresholds of flow-table usage. During online training, the weights are iteratively adjusted accordingly with the aim of making flow-table usage in all switches as close as possible to the desired threshold of flow-table usage and the number of received PacketIn messages at the controller as close as possible to the available queue capacity of the controller. Both models are trained online and both discounts ψ and ϕ are computed using the gradient descent algorithm. An aggregation of both discounts is applied in order to adjust weights of hosts. Aggregation includes max, min and weighted sum functions.

When a weight is adjusted, the controller dictates the new allowable number of flow requests to the host, if any, using the OpenFlow messages *PortStatus* and *PortMod*. Consequently, the controller checks if the host complies with new policies by following changes in its associated weight. If changes remain within an interval $[-\xi, \xi]$, the host is considered in normal behavior, otherwise, it is recognized as compromised and must be blocked. The controller blocks the host by installing a new default-rule within the flow-table by which all new flow requests which are not matched will be immediately dropped. Moreover, the controller evaluates the behavior of all switches and blocks miss-behaving ones. The behavior of a switch is evaluated through a computation of its trustworthiness. Trustworthiness is a measure of cohesiveness and correlation of a switch with its neighborhood and is computed using a weighted version of the correlation degree denoted the Average Nearest-Neighbor Degree (ANND) [8].

Our proposed solution allows to protect both controllers and switches against flooding attacks without introducing new extra components so that no control-plane communication overload will be experienced compared to existing solutions generally and to SDN-GUARD [15] particularly. It consists of anticipating misbehaving nodes and penalizing them accordingly, whether they are benign or malicious nodes. It is worth noting that penalization has no effect on the performance of network and does not lead to the creation of bottlenecks since it is back-propagated to traffic sources. Hence, limiting traffic on sources will contribute to the reduction of TCAM consumption in switches, and so, minimizing the probability of bottlenecks occurrence, especially if a reliable routing scheme is used. Fig. 2 illustrates a workflow diagram of the proposed solution.

4.1.1. Building the training model

In an SDN network, each host h_i is connected to a switch s_j to which it can deliver an amount of new-flow requests r_{h_i} during a period of time t. This amount must not exceed a maximum allowable number of new requests r^h_{max} which is specified by the network operator for efficient resources sharing and to restrict bandwidth exhaustion, so that, $r_{h_i} \leq r^h_{max}$. Besides, each switch s_j can be connected to multiple controllers, as defined in Openflow1.3, but must have only one master controller which replies to its requests. For the same issue of efficient resources management, each switch is licensed to transmit a number of PacketIn messages r_{s_j} which must not exceed a maximum number of PacketIn messages r^s_{max} for a period of time t, so that, $r_{s_i} \leq r^s_{max}$.

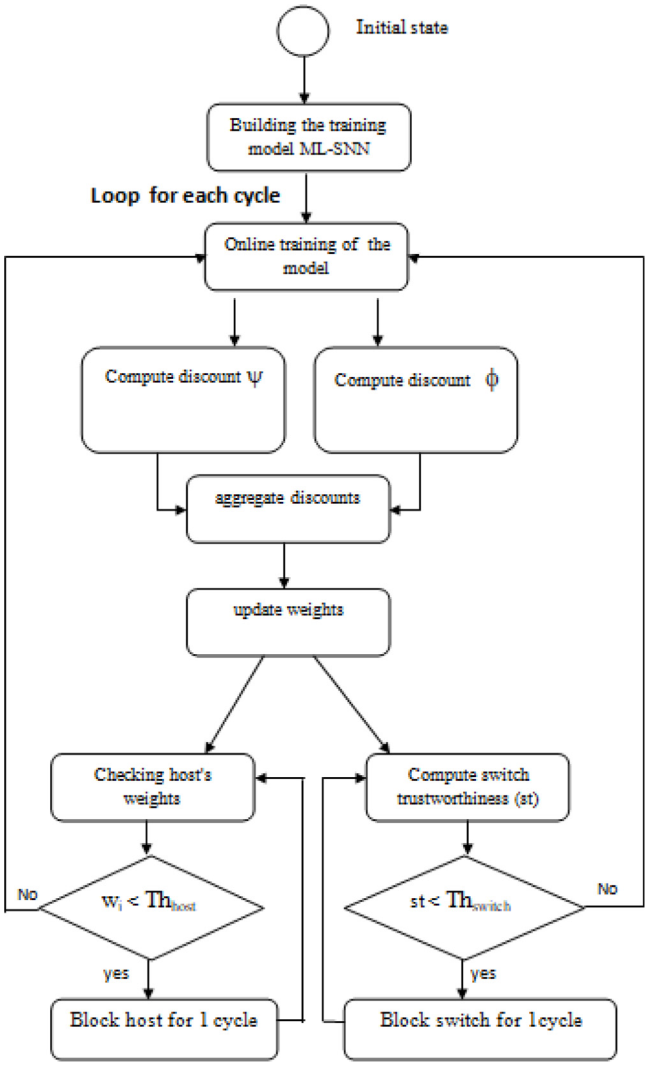

Fig. 2. Overview of the approach.

Given an undirected graph G = (H, S, C) where H denotes the set of hosts, S is the set of edge switches and C is the set of controllers, a weight w^h_{ij} defines the fraction of allowable number of new flows that a host h_i can send to the switch s_j. Besides, a weight w^s_{jk} defines the fraction of allowable number of PacketIn messages that the switch s_j can forward towards its master controller c_k. Given w^h_{ij} and w^s_{jk}, the allowable number of new-flow requests r_{h_i} which is sent by the host h_i towards the switch s_j and the number of PacketIn messages r_{s_j} which is sent by the switch s_j towards the controller c_k are expressed by Eq. (6) and Eq. (7), respectively.

$$r_{h_i} = w^h_{ij} r^h_{max} \tag{6}$$

$$r_{s_j} = w^s_{jk} r^s_{max} \tag{7}$$

where r^h_{max} and r^s_{max} denote the maximum number of new-flow requests and the maximum number of PacketIn messages defined for hosts and switches, respectively, and they are operator-specific. Without loss of generality, we assume that $r^h_{max} = r^s_{max} = r_{max}$ throughout this paper. All weights are in the range of [0, 1], so that, a value of 1 means that the host or the switch can exploit all available resources, while a value of 0 means that sending requests is prohibited.

Hence, an SDN network can be modeled as a Fully Connected Shallow Neural Network (FCSNN), or simply (SNN). SNN consists of one input layer, one output layer and one hidden layer (Fig. 3). Each controller must build its SNN which includes the set of all hosts and all edge switches from its local domain context. The input layer L_0 consists of |H| neurons which represent the set of hosts. The hidden layer L_1 refers to the set of all edge switches connecting hosts to the controller. It contains |S| neurons. Finally, the output layer L_2 consists of one neuron and represents the controller. A weight w^l_{ij}, $\forall l \in \{0, 1\}$, defines the fraction of data that can be transmitted by a neuron n^l_i towards a neuron n^{l+1}_j from the layer, where neuron n^l_i defines the ith neuron from the lth layer.

Weights are learnable parameters of the SNN which are adjusted according to one or more loss functions. When multiple loss functions are considered, the SNN is recognized as a Multi-Loss Shallow Neural Network (ML-SNN) (Fig. 4).

ML-SNN is composed of multiple parallel loss functions in which weights are adjusted according to an aggregation of multiple discounts obtained from backpropagating different losses.

In this paper, two loss functions are considered. The first one is based on the Mean Square Error function and used to evaluate the loss between the available queue capacity of the controller and the predicted number of received PacketIn messages. The *ReLU* activation function is considered in this part. The second one is based on the Cross Entropy (CE) function and used to evaluate the distance of the current usage of flow-table space from the maximum flow-table usage allowed at a switch. The *Softmax* activation function is considered in this section. An algorithm denoted the Parallel Online Deep Learning (PODL) is used to train the ML-SNN and to adjust weights using the

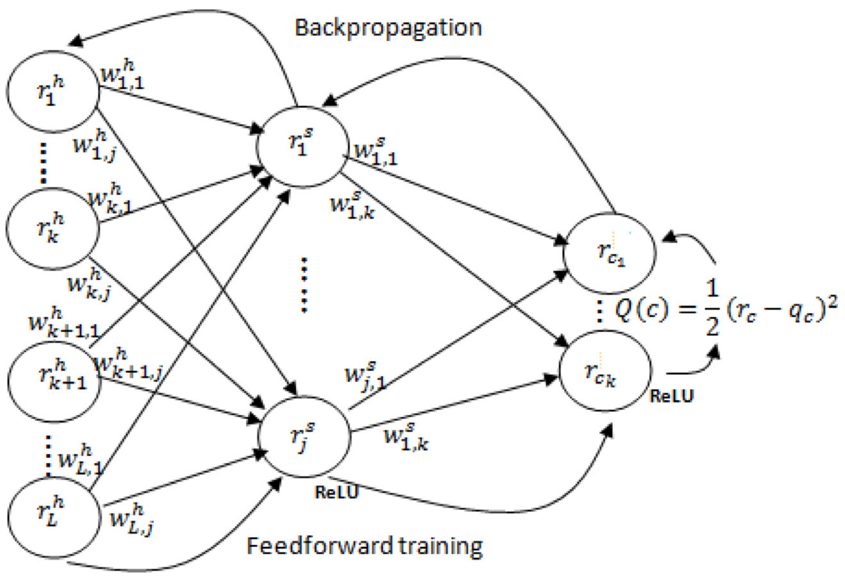

Fig. 3. Training model using Fully Connected SNN.

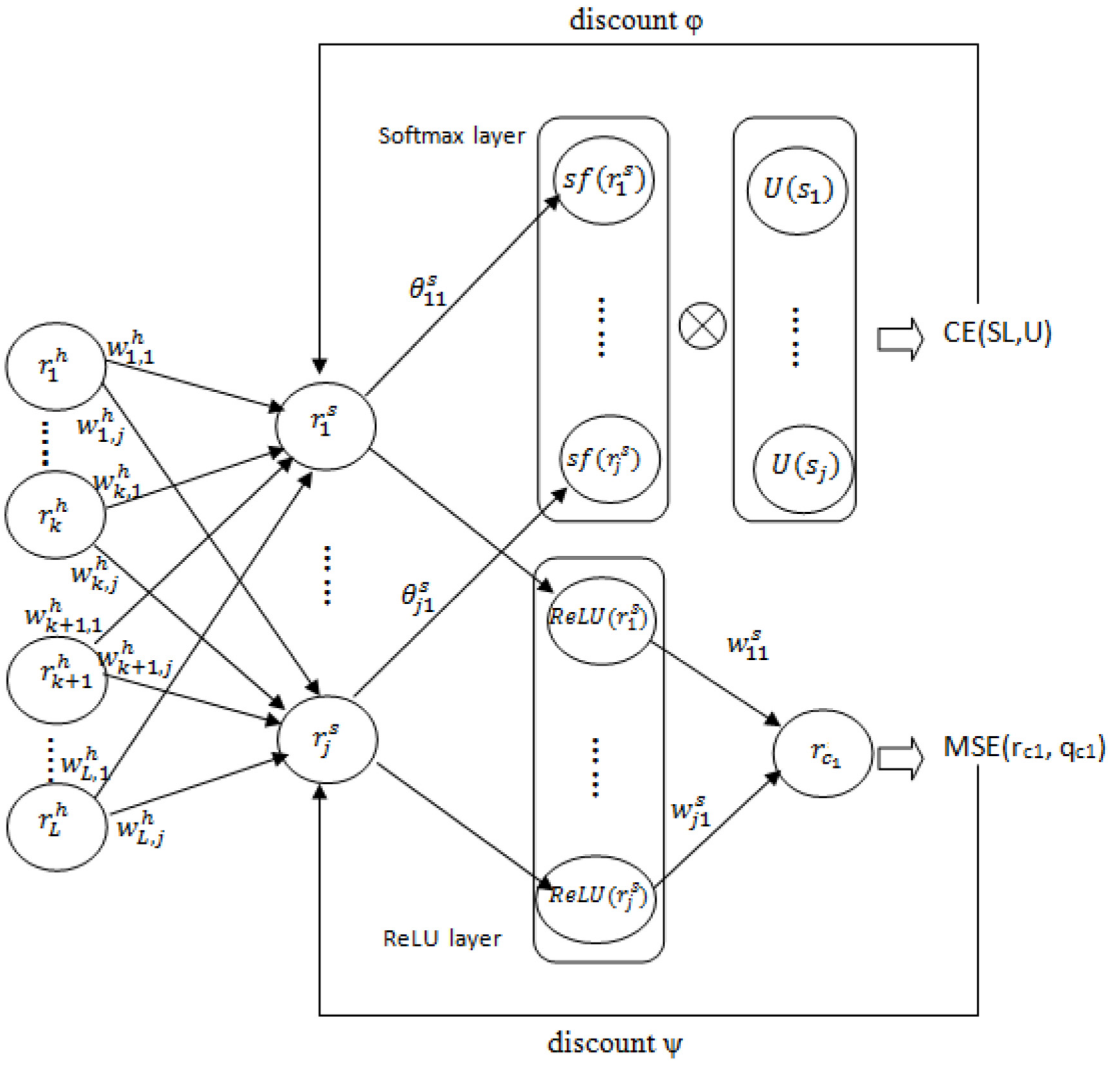

Fig. 4. Multi-Loss Shallow Neural Network (ML-SNN).

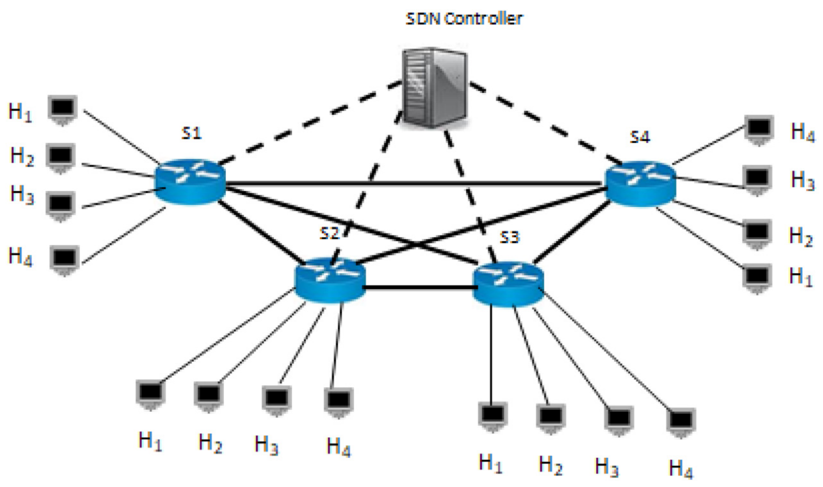

Fig. 5. Experimental Topology.

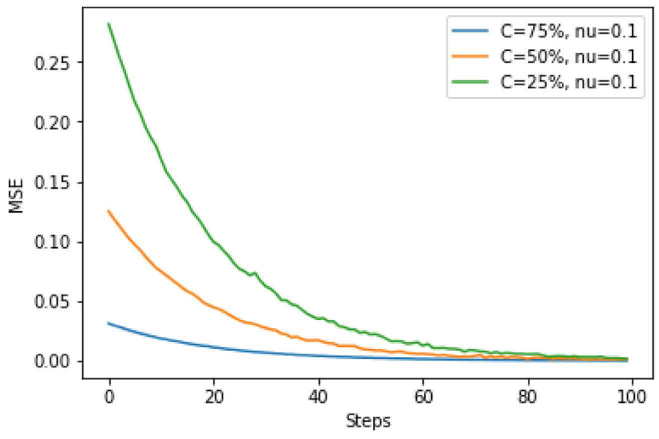

Fig. 6. Mean Square Error Minimization between the predicted number of received PacketIn messages at the controller for fixed queue occupation C ∈ [25%, 50%, 75%] and $\eta = 0.1$.

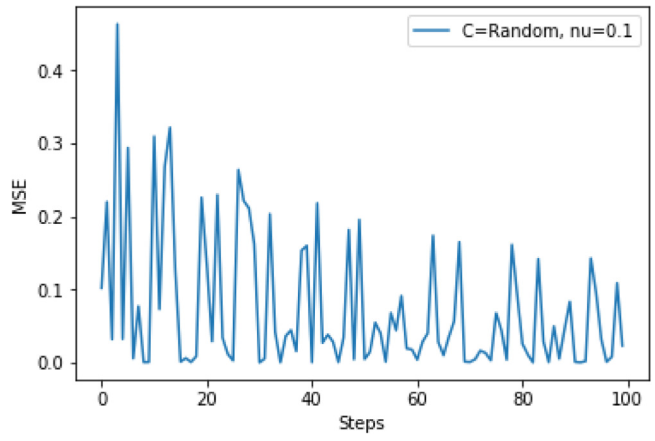

Fig. 7. Mean Square Error Minimization between the predicted number of received PacketIn messages at the controller for random queue occupation.

Gradient Descent algorithm (GD). GD helps to compute discounts ψ_i and ϕ_i associated to a switch s_i, $\forall s_i \in S$.

4.2. The parallel online deep learning algorithm (PODL)

Given a host h_k which is connected to a switch s_i, the weight w^h_{ki} is updated according to two loss functions. Let ψ_i and ϕ_i be the discounts applied to the weight due to the contribution of the number of new flow requests of the host h_k to the queue capacity of the controller c_j and the flow-table usage of the switch s_i, respectively. Therefore, the weight w^h_{ki} is updated as follows (Eq. (8)):

$$w^h_{ki} = w^h_{ki} - \eta \mathbb{A}(\psi_i, \phi_i) r_{h_k} \tag{8}$$

Where η is the step size or the learning rate. $\mathbb{A}()$ is an aggregation function used to fuse both discounts. $\mathbb{A}()$ may be the min, max or the weighted sum function. The discount ϕ_i helps to evaluate the disparity between the current flow-table usage in the switch s_i and the maximum allowable usage, while ψ_i gives information on the contribution of the number of received PacketIn messages from switch s_i to the queue capacity of the controller c_j.

4.2.1. Computing the discount ψ_i of a switch s_i

The discount ψ_i associated to a switch s_i which is connected to a controller c_j is computed based on the weight w^s_{ij}. To update the weight w^s_{ij}, the backpropagation process is used. During the backpropagation, the new weight is computed through the gradient descent algorithm as follows (Eq. (9)):

$$w^s_{ij} = w^s_{ij} - \eta \frac{\partial Q_j}{\partial w^s_{ij}} \tag{9}$$

Where $Q_j = \frac{1}{2}(r_{c_j} - q_{c_j})^2$ is the Mean Square Error (MSE) loss function, η is the step size or the learning rate, q_{c_j} denotes the available queue capacity and r_{c_j} is the number of received PacketIn messages in the controller c_j. The partial derivative $\frac{\partial Q_j}{\partial w^s_{ij}}$ is computed through the chain rule as presented in Eq. (10).

$$\frac{\partial Q_j}{\partial w^s_{ij}} = \frac{\partial Q_j}{\partial r_{c_j}} \frac{\partial r_{c_j}}{\partial w^s_{ij}} = (r_{c_j} - q_{c_j}) r_{s_i} \tag{10}$$

In the same way, the weight w^h_{ki} between a host h_k and a switch s_i can be adjusted as follows (Eq. (11)):

$$w^h_{ki} = w^h_{ki} - \eta \frac{\partial Q_j}{\partial w^h_{ki}} \tag{11}$$

Using the chain rule, the partial derivation $\frac{\partial Q_j}{\partial w^h_{ki}}$ can be expressed as follows (Eq. (12)):

$$\frac{\partial Q_j}{\partial w^h_{ki}} = \frac{\partial Q_j}{\partial r_{c_j}} \frac{\partial r_{c_j}}{\partial r_{s_i}} \frac{\partial r_{s_i}}{\partial w^h_{ki}} \tag{12}$$

Or alternatively (Eq. (13));

$$\frac{\partial Q_j}{\partial w^h_{ki}} = (r_{c_j} - q_{c_j}) w^s_{ij} r_{h_k} \tag{13}$$

So, the discount ψ_i is computed as follows (Eq. (14)):

$$\psi_i = (r_{c_j} - q_{c_j}) w^s_{ij} \tag{14}$$

4.2.2. Computing the discount ϕ_i of a switch s_i

A *softmax* layer is added to the previous model so that each neuron in the hidden layer is connected to a node from the *softmax* layer using a weighted link with θ parameter. Therefore, the Cross-Entropy(CE) loss function is used to assess if the maximum usage of flow-tables in switches is respected on the switch, otherwise, weights of hosts must be adjusted in order to reduce the CE loss.

Cross entropy indicates the distance between what the model believes the output distribution should be, and what the original distribution really is. It is defined as $H(y,p) = -\sum_i y_i \log(p_i)$, where $y = (y_1, \dots, y_{|S|})$ defines the expected distribution and $p = (p_1 \cdots, p_{|S|})$ is the resulting distribution. Thereby, the *softmax* layer is used to convert the number of PacketIn messages of all switches into a distribution of probabilities representing the usage of flow-tables in all switches.

Cross-entropy loss function is a widely used alternative of squared error. It is used when node activation can be understood as representing the probability that each hypothesis might be true, i.e. when the output is a probability distribution. Thus, it is used as a loss function in neural networks which have *softmax* activation function in the output layer. Both discounts are computed using the online backpropagation algorithm (Fig. 4).

Let θ_{ij} be a learning parameter associated to the switch s_i which is connected to the controller c_j and let f_{ij} and F_j be the output of the *softmax* function on r_{s_i} and the result of the cross entropy loss function, respectively. The two outputs are defined as follows:

(i) $F_j = CE(f_{ij}, U^j_{s_i})$
(ii) $f_{ij} = softmax(r_{s_i}\theta_{ij})$
(iii) $r_{s_i} = \sum_{h_k \in H(s_i)} w^h_{ki} r_{h_k}$

Where $U^j_{s_i} \in [0,1]$ denotes the threshold of flow-table usage in the switch s_i. Let denote $\overrightarrow{f_j} = (f_{ij})_{\forall s_i \in S(c_j)}$ the distribution of probability obtained from the softmax layer and $\overrightarrow{U^j} = (U^j_{s_i})_{\forall s_i \in S(c_j)}$ the set of thresholds of flow-table usage for all switches connected to the controller c_j. Therefore, the cross entropy $F_j = CE(\overrightarrow{f_j}, \overrightarrow{U^j})$ between $\overrightarrow{f_j}$ and $\overrightarrow{U^j}$ is computed as follows (Eq. (15)):

$$F_j = -\sum_{s_i \in S(c_j)} U^j_{s_i} \log(f_{ij}) \tag{15}$$

The discount ϕ_i associated to a switch s_i which is connected to a controller c_j is computed based on the parameter θ_{ij}. To update θ_{ij}, we use the gradient descent based backpropagation algorithm. The parameter θ_{ij} is adjusted as follows (Eq. (16)):

$$\theta_{ij} = \theta_{ij} - \eta \frac{\partial F_j}{\partial \theta_{ij}} \tag{16}$$

Where $\frac{\partial F_j}{\partial \theta_{ij}}$ is computed as follows (Eq. (17)):

$$\begin{aligned}\frac{\partial F_j}{\partial \theta_{ij}} &= \frac{\partial F_j}{\partial f_{ij}} \frac{\partial f_{ij}}{\partial \theta_{ij}} \\ &= (f_{ij} - U_{s_i}) r_{s_i}\end{aligned} \tag{17}$$

Moreover, the weight w^h_{ki} between a host h_k and a switch s_i can be adjusted as follows (Eq. (18)):

$$w^h_{ki} = w^h_{ki} - \eta \frac{\partial F_j}{\partial w^h_{ki}} \tag{18}$$

The partial derivation $\frac{\partial F_j}{\partial w^h_{ki}}$ can be computed as follows (Eq. (19)):

$$\begin{aligned}\frac{\partial F_j}{\partial w^h_{ki}} &= \frac{\partial F_j}{\partial r_{s_i}} \frac{\partial r_{s_i}}{\partial w^h_{ki}} \\ &= \psi_i r_{h_j}\end{aligned} \tag{19}$$

Therefore, ϕ_i is expressed as follows (Eq. (20)):

$$\begin{aligned}\phi_i &= \frac{\partial F_j}{\partial r_{s_i}} \\ &= \frac{\partial F_j}{\partial f_{ij}} \frac{\partial f_{ij}}{\partial r_{s_i}} \\ &= (f_{ij} - U_{s_i})\theta_{ij}\end{aligned} \tag{20}$$

The algorithm **PODL** (Algorithm 1) outlines the proposed solution for updating weights of all hosts taking into account both the available memory space of controllers and switches.

```
Algorithm 1: PODL($c_k$)
   Data: $\{r_{h_j}\}\forall h_j$, $\{U_{s_i}\}\forall s_i$, $r_{max}$, $q_{c_k}$, $\eta$
   Result: $W^h = \{w^h_{ji}\}$, $\forall h_j \in H(s_i) | s_i \in S(c_k)$
 1 for $s_i \in S(c_k)$ do
 2     Compute $r_{s_i}$
 3 Compute $r_{c_k}$
 4 do in parallel
 5     for $s_i \in S(c_k)$ do
 6         Compute_discount_$\psi(s_i)$
 7         Compute_discount_$\phi(s_i)$
 8 for $s_i \in S(c_k)$ do
 9     for $h_j \in H(s_i)$ do
10         $w^h_{ji} \leftarrow w^h_{ji} - \eta\Gamma(\psi_{s_i}, \phi_{s_i}) r_{h_j}$ : using equation (8)
11 return $W^h$
```

The routines $Compute\_discount\_\psi()$ and $Compute\_discount\_\phi()$ are described in algorithm Eq. (18) and algorithm 3, respectively.

```
Algorithm 2: Compute_discount_$\psi(s_i)$
   Data: $r_{c_k}$, $q_{c_k}$, $r_{s_i}$, $w^s_{ik}$
   Result: $\psi(s_i)$
 1 $w^s_{ik} \leftarrow w^s_{ik} - (r_{c_k} - q_{c_k}) r_{s_i}$: using equations (9) and (10)
 2 $\psi(s_i) \leftarrow (r_{c_k} - q_{c_k}) w^s_{ik}$ : using equation (14)
 3 return $\psi(s_i)$
```

```
Algorithm 3: Compute_discount_$\phi(s_i)$
   Data: $r_{s_i}$ , $U_{s_i}$, $\theta_{ik}$, $\eta$
   Result: $\phi(s_i)$
 1 $f_{ik} \leftarrow$ softmax$(\theta_{ik} r_{s_i})$
 2 $\theta_{ik} \leftarrow \theta_{ik} - \eta(f_{ik} - U_{s_i}) r_{s_i}$ : using equation (16)
 3 $\phi(s_i) = (f_{ik} - U_{s_i})\theta_{ik}$: using equation (20)
 4 return $\phi(s_i)$
```

5. Trustworthiness inspection

The controller inspects periodically the behavior of each host h_i by evaluating the evolution of the associated weight w^h_{ij} throughout the time. As this weight falls below a threshold th_h, the controller inflicts to the switch to block the host h_i by prohibiting reception of new flow requests from p_{h_i}. Hence, the controller installs a default flow-entry within the flow-table of the switch that drops any packet arriving from h_i and to which no match was found. The host can no longer send new flow requests for a period of time. The period of time is defined through the Idle-Timeout field and is discussed later.

The controller must also monitor the behavior of each switch by evaluating its trustworthiness among its neighborhood. Generally, trustworthiness is defined as the confidence that a node i has on node j about how the latter will perform as expected. Trustworthiness is associated to security, efficiency and reliability. In this work, a node is said to be trusted if it is reliable. Reliability is defined as the quality of performing consistently well.

Practically, the local clustering coefficient [8] is used to evaluate the reliability of a switch. The clustering coefficient defines a density measure of local connections and it is an indication of its embeddedness. So that, more the coefficient is going down, more the connections density is low and more the device is considered unreliable. The clustering coefficient is computed through the Average Nearest-Neighbor Degree (ANND) [51]. ANND is a metric of cohesiveness and correlation which can be used in accordance with the clustering coefficient. Given a switch s_i, The ANND, denoted Γ_{s_i}, is expressed as follows (Eq. (21)):

$$\Gamma^{w}_{s_i} = \frac{1}{g_{s_i}} \sum_{v_j \in N_{s_i}} w^{\langle s,h \rangle}_{ij} deg_j \tag{21}$$

Where $g_{s_i} = \sum_{v_j \in N_i} w^{\langle c,h \rangle}_{ji}$ defines the switch strength and deg_i is the degree of the switch s_i or the host h_i. As $\Gamma^{w}_{s_i}$ of the switch s_i falls below a threshold th_s, the switch is considered as compromised and it is no longer authorized to forward PacketIn messages to the controller. Hence, the controller must transmit a *PortMod* message towards the switch in which the feature *NoPacketIn* is activated.

The algorithm **TrustDevice** (Algorithm 4) outlines the proposed scheme for trustworthiness inspection. The timeout of the default rule can be fixed for a random duration, or set to the most popular duration of DoS attacks, as published by researches.

Algorithm 4: TrustDevice(c_m)

```
1  call PODL($c_m$)
2  for $s_j \in S(c_m)$ do
3      for $h_k \in H(s_j)$ do
4          if $w^h_{kj} \leq th_h$ then
5              install defaultRule($s_j, h_k$, Drop)
6              $H(s_j) = H(s_j) - \{h_k\}$
7  compute $\Gamma^w_{s_j}$ : using equation (21)
8  if $\Gamma_{s_j} \leq th_s$ then
9      Send PortMod($s_j$, NoPktIn=true)
10     $S(c_m) = S(c_m) - \{s_j\}$
```

6. Performance evaluation of algorithms

In this section, we evaluate the performance of our proposed approach. First, we study the effectiveness of the PODL algorithm in mitigating flooding-based DDoS attacks. Second, we analyze the capacity of each algorithm to ensure steady queue capacity in the controller and not to-be-overloaded memory state in switches.

6.1. Experimental settings

The experimental topology is presented in Fig. 5. It consists of one controller, n switches and k hosts per each switch (n = 4, k = 4). The queue occupation of the controller varies in the set [25%, 50%, 75%]. Initially, inputs of all hosts are randomly distributed in the range [0, r_{max}].

We run algorithms 100 times (nsteps = 100) without considering a DoS attack case, then, we rerun experiments another time and we enforce the input of the host (Host1) to 90% of the maximum number of allowable number of new flow requests in order to simulate a DoS attack during 50 steps. All algorithms are implemented using Python3 [52] and simulated through the platform JupyterLab [53].

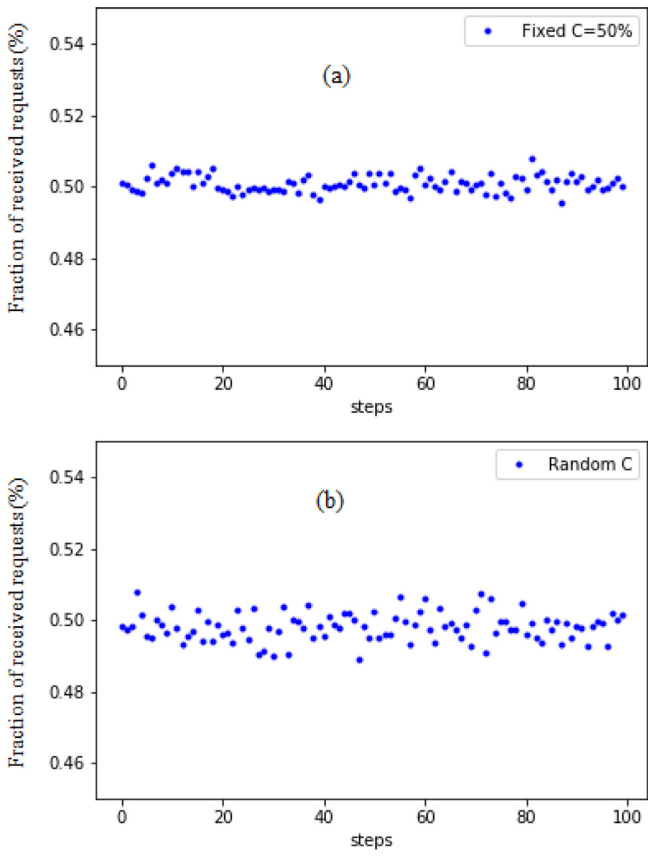

Fig. 8. Model Validation: number of received PacketIn message when using (a) fixed queue occupation, (b) random queue occupation.

6.2. ODL performance evaluation

Fig. 6 illustrates the evaluation of the disparity between the predicted number of received PacketIn messages and the queue capacity of the controller. Clearly, the loss looks to be minimized as long as the model is trained.

As shown in results, the network converges to a stable state quickly and the queue capacity is closely respected (MSE≈0). For the queue occupation of $C = 75\%$, the optimized state is reached after 40 steps, while it is 60 steps when $C = 50\%$ and more than 80 steps for $C = 25\%$. Moreover, the disparity between the queue capacity and the number of received PacketIn messages has been evaluated in case when the queue occupation C varies randomly in the range [0, 1]. As shown in Fig. 7, the model is incrementally trained which leads to a minimized loss function less than 9%.

To validate the proposed learning algorithm, weights resulting after the training are validated through a Dataset of samples. For a queue occupation of $C = 50\%$ and then for a random queue occupation, we measure the number of received PacketIn messages at the controller considering a random distribution of the number of new flow requests for all hosts. Results are presented in Fig. 8.

In both cases, the number of received PacketIn messages at the controller is maintained close to the queue size defined through a queue occupation of $C = 50\%$. PODL is able to enforce respecting the available queue size of the controller even when the available queue size is randomly changing between steps.

Finally, we evaluate the changes on host weights along the learning process. We run a first simulation without DoS attack attempts, then, we rerun the same simulation considering a DoS attack attempt in the host Host1. We consider also various queue occupation from the set [25%, 50%, 75%]. As shown in Fig. 9, the weight associated to the host

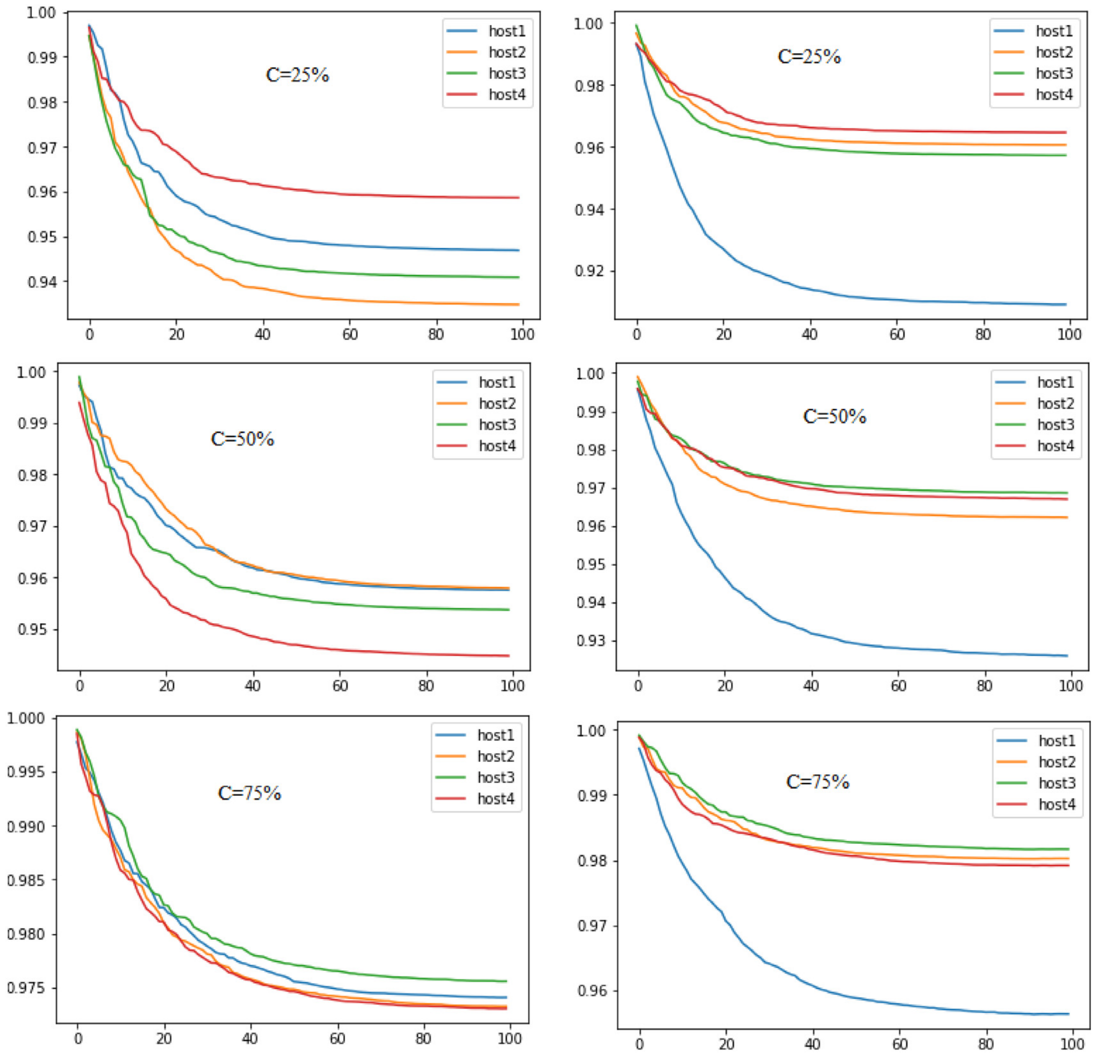

Fig. 9. Weights update for hosts considering various fixed queue occupations considering normal and abnormal behaviors of host H1.

Host1 is largely decreased compared to other hosts since it behaves as a compromising host. Weights of all Other hosts are scarcely affected by the DoS attack attempt in Host1. The host which is the source of a DoS attack is penalized widely while other hosts are influenced barely. The proposed solution is able to react properly even in a distributed DoS attack case. In this case, each host will be penalized proportionally to its contribution to the number of received PacketIn messages at the controller. Fig. 10 illustrates the variation of the weights of hosts when, simultaneously, a DDoS attack on the controller takes place.

In experiment (a), the attack is launch by host h1 and host h2. In experiment (b), host h3 participates in the attack with h1 and h2. Clearly, hosts participating in the attack (host1 and host2) are more penalized and have a large discount in their weights compared to hosts which are well-behaving (host3 and host4). As the host host3 participates in the DDoS attack, it is weight is more penalized in the same way as done with hosts host1 and host2. However, the weight of host4 remains stable since the host does not participate in the attack.

Finally, the simulation aims at evaluating the number of received new flow requests for each switch compared to the current flow-table usage. Values of flow-table usage are randomly chosen in the range [0, 1]. Results are shown in Fig. 11, switches s_1, s_6 and s_7 are overloaded since the resulting number of new flow requests (* *marks*) exceeds the threshold of flow-table usage (*x marks*). the ODL algorithm cannot avoid flow-table flooding since no control is applied on the flow-table occupation of each switch, which leads to unprecedented switch memory overwhelming.

To avoid being flooded, switches which are going to be overloaded must be penalized more than those which are not. The PODL is able to meet this objective through the ϕ discount.

6.3. PODL performance evaluation

The first simulation aims to illustrate the contribution of the PODL algorithm in terms of maintaining optimized queue size in the controller and optimized flow-table usage in each switch.

As shown in Fig. 12, the distribution of the number of new flow requests is getting closer to the available flow-table spaces in switches. The cross entropy loss is minimized as soon as more samples are used for training. Consequently, the difference between predicted number of received PacketIn messages and the available queue size is decreasing and reaches less than 3e-8.

Fig. 13 highlights the impact of the PODL algorithm on the performance of the switch s_1 regarding the flow-table usage. We define the metric $\Delta_{s_i} = U_{s_i} - r_{s_i}$ as the difference between the available flow-table space and the number of new flow requests received at the switch s_i. As shown in results, Δ_{s_1} seems to be canceled as the network is trained which means that the number of new flow requests received at the switch s_1 is moving to be steadily equal to the available flow-table space of the switch.

The PODL algorithm performs better than the ODL algorithm. This is exhibited in Fig. 14. Clearly, resulting amount of new flow requests

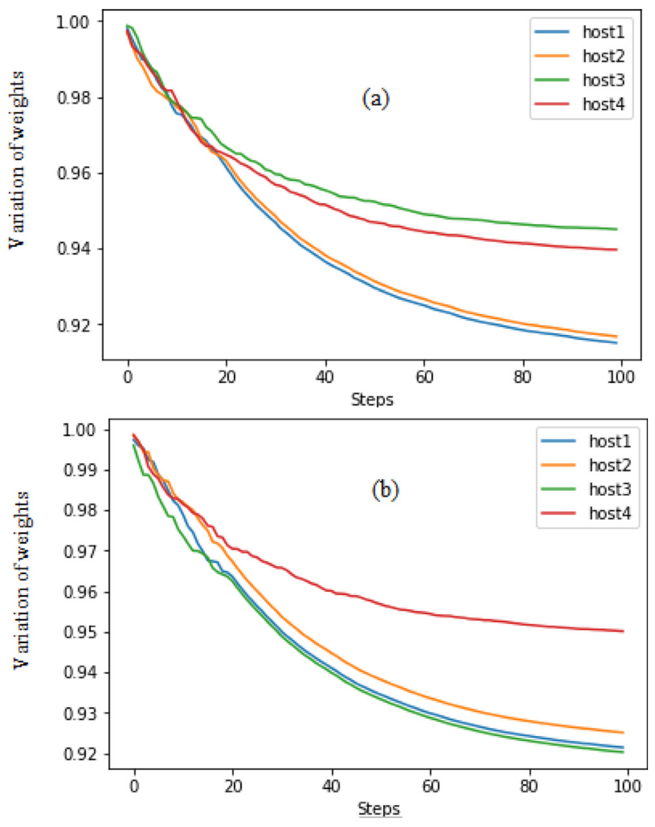

Fig. 10. Weights updating for hosts with fixed queue size taking in account Distributed DoS attack: (a) attack initiated by h1 and h2, (b) attack initiated by h1, h2 and h3.

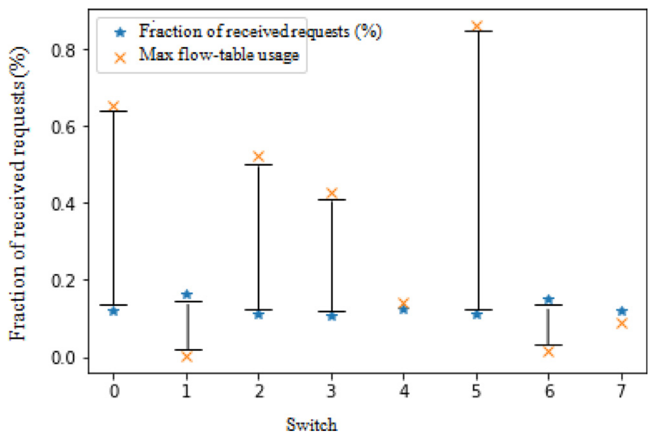

Fig. 11. Switch's flow-table usage using the ODL algorithm.

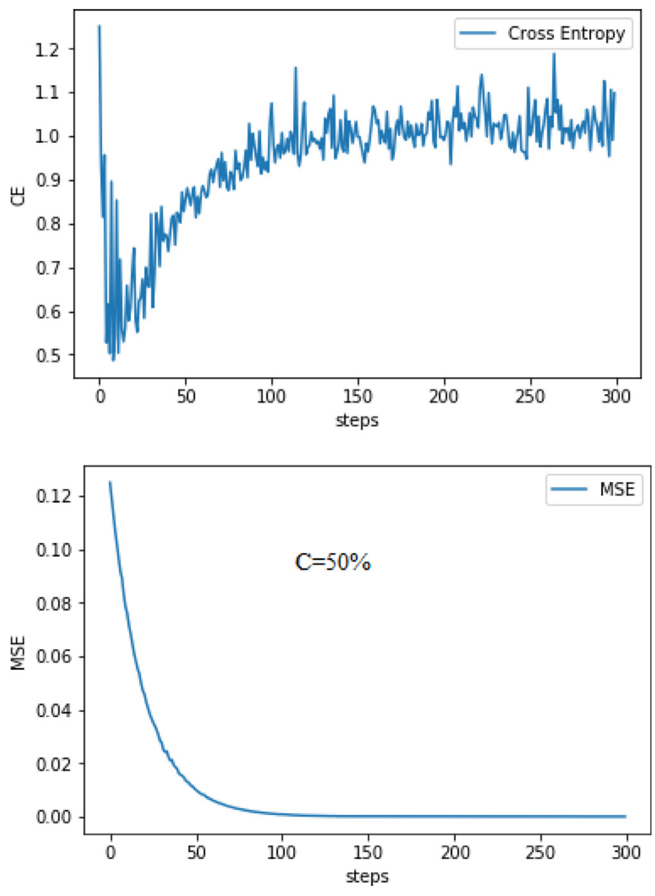

Fig. 12. Impact of the PODL on: (a) Cross-Entropy loss (CEE) (b) Mean Square Error(MSE) loss.

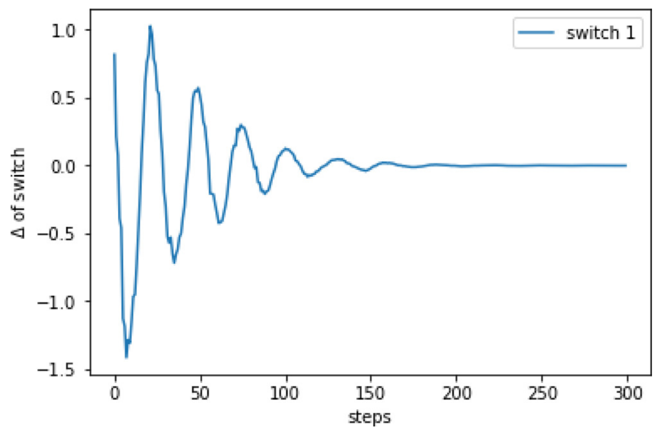

Fig. 13. Evolution of Δ metric for the switch S_1.

received in each switch remains below the available flow-table space. Hence, flow-tables are appropriately filled without exceeding available capacities neither underusing the allowed space (this is the case of switches 3 and 4).

Finally, we evaluate the impact of TrustDevice algorithm on host's weights. In this simulation, we suppose that hosts h1 and h2 initiate a DDoS attack. At t=0, hosts inundate the network with a growing number of new-flow requests for a period of time (100 steps). We run the simulation for 1000 steps. Fig. 15 illustrates the variation of weights. As the controller inspects weights of all hosts, it takes a decision to block hosts h1 and h2 forthwith (Algorithm TrustDevice). Consequently, weights of h3 and h4 increase since they are considered as benign hosts so that they capitalize on resources released from h1 and h2 when they are blocked, so, they acquire more resources and profit from a higher throughput.

7. Efficiency of the proposed approach

In this second part of the experimental section, the efficiency of the overall approach is analyzed. Mainly, the rate of received PacketIn messages at the controller and the acceptance rate of new requests are evaluated. Results are compared with the case of no rate-limiting (no-RL), the case of fixed-window rate-limiting mechanism (fw-RL) (a window of 5kflows/s) and the case of sliding-window rate-limiting mechanism (sw-RL) with the same window size.

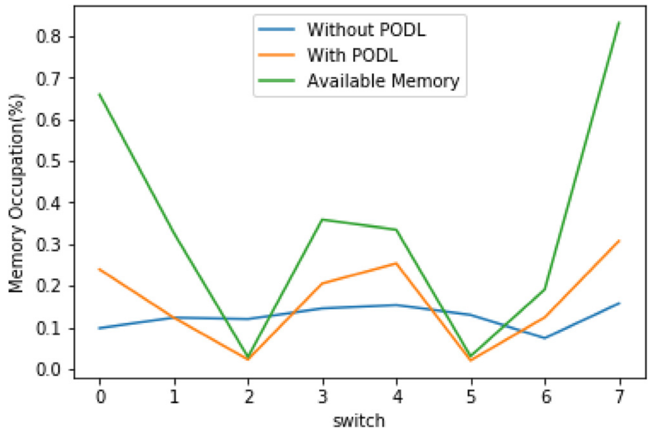

Fig. 14. Switch flow-table usage using the PODL algorithm.

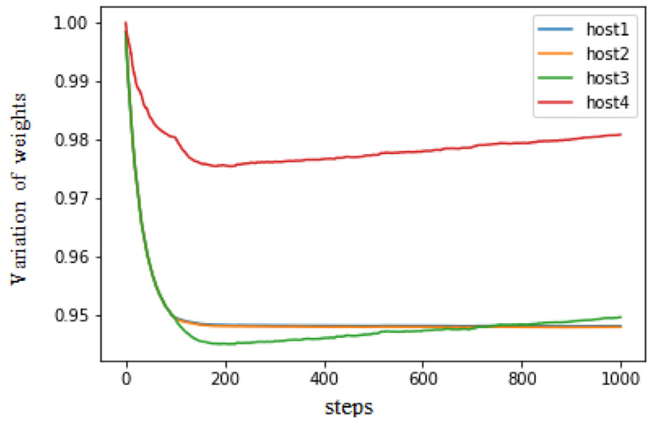

Fig. 15. Impact of the algorithm TrustDevice on host weights.

7.1. Experimental settings

We consider the topology described in Fig. 16. All algorithms are implemented in the floodlight controller [54], which is a java-based OpenFlow controller. In this simulation, one controller, 10 switches and 6 hosts are considered. The controller is able to handle PacketIn messages at the rate of 80 kfps (flow per second).

Each host is initially allowed to generate 15 kfps (flow per second) towards the target server h_5. After a period of 40 s, hosts h_3 and h_4 start increasing their throughput until reaching r_{max} = 40 kfps, then, they start decreasing it until getting back to their initial rates. This phenomena takes over 30 s during which attackers are generating new TCP flows. The DoS attack is accomplished using *hping3* network tool [55] which floods the target server with a large number of TCP-SYN packets using different IP source addresses. Data plane is simulated using Mininet [56] and all switches use OpenVSwitch [57]. We consider the **MAX** function to aggregate discounts.

7.2. Simulation results

The first simulation aims at studying the PacketIn arrival rate at the floodlight controller. Results are shown in Fig. 17.

The behavior of the controller seems to be normal until the attack takes place. The number of incoming requests increases incrementally. It reaches the maximum allowable number of requests of the controller for both classical schemes, fw-RL and sw-RL, while it remains under this limit for our proposed mechanism. Compared to no-RL and fw-RL, the proposed mechanism succeeds in avoiding the inundation of the controller by preventively reducing the number of received PacketIn messages up to 31%. This is basically caused by the fact that our approach can preemptively limit the rate of each host proportionally to its contribution to the incoming traffic of requests surging at the controller. When no rate-limiting logic is implemented in the controller, the capacity of the controller is exhausted leading to quick breakdown of the service (at t = 70 s).

The second simulation aims at analyzing the number of requests sent by the host h3 compared to the number of accepted ones (Fig. 18). In order to simulate a DoS attack, h3 increases incrementally its number of new requests towards the switch over the time. Thus, the switch transmits all requests to the controller.

As the number of requests increases, the controller reacts immediately by limiting the number of accepted PacketIn. This limitation is proportional to the number of new requests transmitted by each host. Hence, the controller starts dropping incoming PacketIn associated to requests from the host h3. As this host continues sending more new requests, the controller continues decreasing the number of accepted PacketIn from h3 until reaching a fixed threshold th_h. At this point, the host h3 is blocked and thereby forbidden to send new requests. Decision on blocking a host is made by the controller and transmitted to the correspondent switch through a new flow-entry which drops any new packet which has not been matched. Clearly, the DoS attack fails to overload the controller since the controller anticipates the occurrence of such phenomena and reacts to changes in the host's behavior in short time.

Fig. 19 illustrates the reaction of our approach with a host having a legitimate activity (benign host). At the beginning, all requests are served without loss. Later, the host h0 starts increasing the number of new flow requests but for a short period of time. Consequently, the number of accepted requests is implicitly reduced as response to the short growth of the number of new requests sent by host h0. More later, the host h0 decreases the number of new requests and resumes its normal behavior. Therefore, the controller does not block the host h0 and the number of accepted requests increases another time reactively until resuming the first rate. Contrarily to the host h3, the host h0 was recognized as a legitimate node. As shown in Fig. 19, the reaction of the controller is not right away in order to give time to the host to resume its normal behavior (the duration is about 10 s before the controller starts decreasing the number of accepted requests). The proposed approach does not classify a node as a malicious device until it persists increasing its number of new requests despite policies inflicted by the controller.

The reaction of the controller to any miss-behaving host is done for each host independently. As shown in Fig. 20, the number of accepted requests changes inversely proportional to the number of transmitted new requests.

8. Conclusion

This paper deals with the issue of the (D)Dos attack mitigation on SDN networks. Mainly, the mitigation process is based on a sliding window rate limiting logic which uses online learning to train an Multi-Loss SNN (ML-SNN) model in order to ensure that the controller queue capacity and the available flow-table space of switches are never exceeded.

Basically, network training is based on the Online Deep Learning algorithm (ODL) which, Unfortunately, can only train the network regarding one constraint while our problem focuses on two constraints: (i) the available queue capacity of the controller and (ii) the available flow-table space of each switch.

Therefore, a new algorithm denoted Parallel Online Deep Learning (PODL) is proposed. It consists of computing two discounts by running two parallel training models for the constraints and then updating learnable weights using a merged value of those discounts.

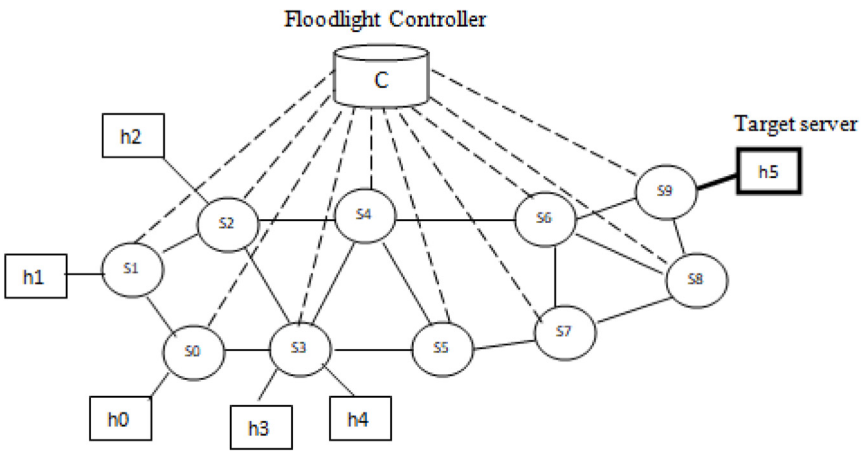

Fig. 16. Simulation topology.

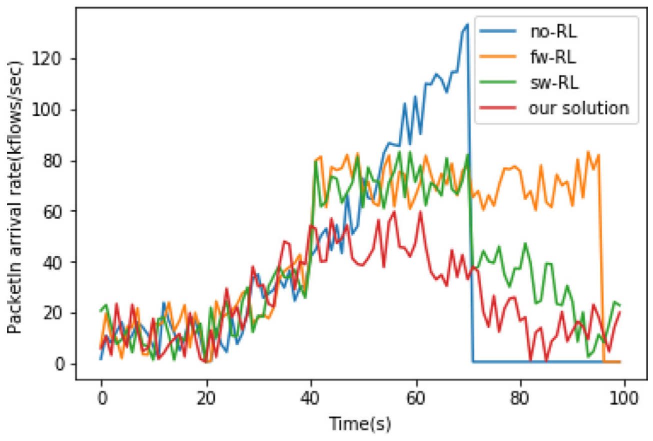

Fig. 17. PacketIn arrival rate at the controller.

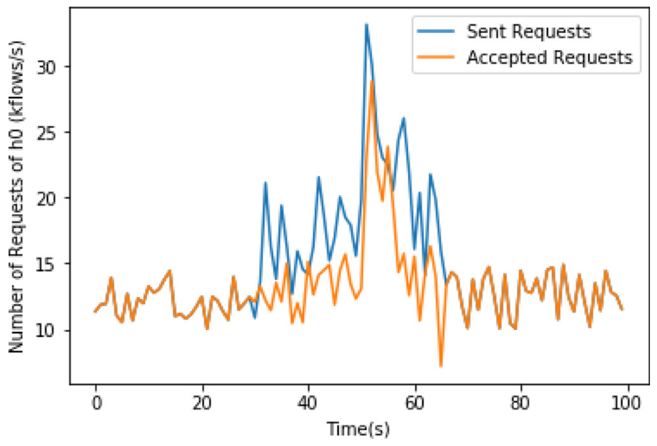

Fig. 19. Number of sent requests vs number of accepted requests of host h0 (benign host).

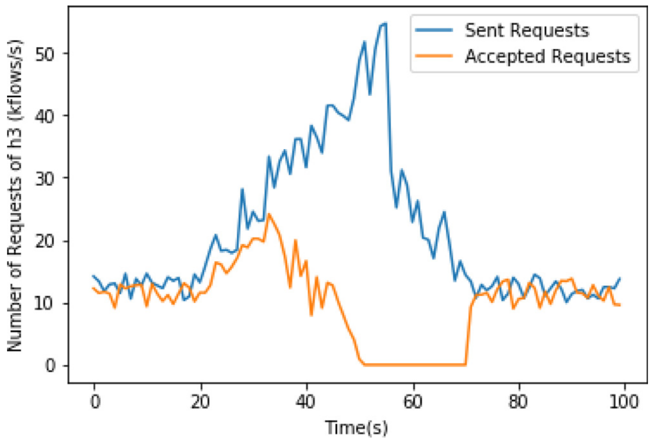

Fig. 18. Number of Sent requests vs number of accepted requests of host h3 (malicious host).

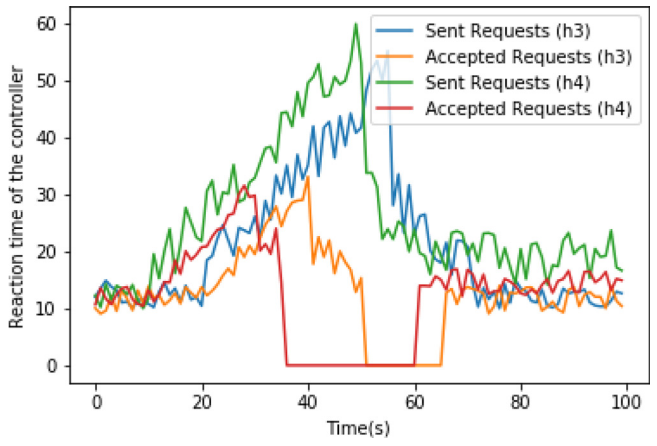

Fig. 20. Reaction points of the controller for two different rates of transmitted requests (h3 and h4).

As shown in results, PODL performs better than ODL in terms of respecting the controller queue capacity and reducing the risk of burdening flow-tables. The host which the weight goes under a fixed threshold is considered as a compromising entity and is immediately prohibited to send new-flow requests. Similarly, a switch which the degree of correlation falls below a fixed threshold is considered distrusted and so it can no longer send PacketIn messages to the controller. Malicious switches and hosts remain in blocked mode unless they retrieve their normal behavior.

In future work, the application of this approach in large scale networks will be addressed. Hence, flat and hierarchical SDN architectures holding multiple controllers will be focused in order to prove the efficiency of the proposed solution in mitigating DDoS attacks on distributed environments. Moreover, more concerns will be raised to improve the trustworthiness evaluation of switches using a combination of the Dempster–Shafer (D–S) evidence theory and the fuzzy logic.

CRediT authorship contribution statement

Ali El Kamel: Conception and design of study, Acquisition of data, Analysis and/or interpretation of data, Writing – original draft. **Hamdi Eltaief:** Conception and design of study, Analysis and/or interpretation of data. **Habib Youssef:** Conception and design of study, Revising the manuscript critically for important intellectual content.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] D. Kreutz, F.M.V. Ramos, P. Esteves Verissimo, et al., Software-defined networking: A comprehensive survey, Proc. IEEE 103 (2014) 10–13.
[2] Big switch networks, the open SDN architecture, 2012, http://www.bigswitch.com/sites/default/files/sdn_overview.pdf.
[3] Nick McKeown, et al., OpenFlow: enabling innovation in campus networks, ACM SIGCOMM CCR, Vol. 38 no. 2, 2008.
[4] Seungwon Shin, Guofei Gu, Attacking software-defined networks: A first feasibility study, in: HotSDN13, 2013.
[5] I. Ahmad, S. Namal, M. Ylianttila, A. Gurtov, Security in software defined networks: a survey, IEEE Commun. Surv. Tutor. 17 (4) (2015) 2317–2346.
[6] R. Cohen, L. Lewin-Eytan, J.S. Naor, et al., On the effect of forwarding table size on SDN network utilization, in: INFOCOM IEEE, 2014, pp. 1734-1742.
[7] Doyen Sahoo, Quang Pham, Jing Lu, Steven C.H. Hoi, Online deep learning: Learning deep neural networks on the fly, in: Proceedings of IJCAI-18, juillet 2018, 2018, pp. 2660–2666, http://dx.doi.org/10.24963/ijcai.2018/369.
[8] D. Yao, P. van der Hoorn, N. Litvak, Average nearest neighbor degrees in scale-free networks, J. Int. Math. (2018) http://dx.doi.org/10.24166/im.02.2018.
[9] G. Sezer S. Scott-Hayward, SDN security: A survey, in: IEEE Conference on Network Softwarization, NetSoft, 2013, pp. 1–7.
[10] S. Shin, G. Gu, Attacking software-defined networks: A first feasibility study, in: ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, 2013, pp. 165–166.
[11] R. Kandoi, M. Antikainen, Denial-of-service attacks in OpenFlow SDN networks, in: IFIP/IEEE International Symposium on Integrated Network Management, IM, 2015, pp. 1322–1326.
[12] L. Wei, C. Fung, FlowRanger: A request prioritizing algorithm for controller DoS attacks in software defined networks, in: IEEE International Conference on Communications, ICC, 2015, pp. 5254–5259.
[13] R. Sahay, G. Blanc, Z. Zhang, H. Debar, Towards autonomic DDoS mitigation using software defined networking, in: Network and Distributed System Security (NDSS) Symposium, 2015.
[14] N.-N. Dao, J. Park, M. Park, S. Cho, A feasible method to combat against DDoS attack in SDN network, in: International Conference on Information Networking, ICOIN, 2015, pp. 309–311..
[15] Dridi Lobna, Mohamed Faten Zhani, SDN-guard: DoS attacks mitigation in SDN networks, 2016, http://dx.doi.org/10.1109/CloudNet.2016.9.
[16] L. Yang, H. Zhao, DDoS attack identification and defense using SDN based on machine learning method, in: 15th International Symposium on Pervasive Systems, Algorithms and Networks, I-SPAN, 2018, pp. 174–178, http://dx.doi.org/10.1109/I-SPAN.2018.00036.
[17] Dong. Li, et al., Using SVM to detect DDoS attack in SDN network, IOP Conf. Ser. Mater. Sci. Eng. 466 (2018) 012003, http://dx.doi.org/10.1088/1757-899X/466/1/012003.
[18] M.S. Elsayed, N.-A. Le-Khac, S. Dev, A.D. Jurcut, Machine-learning techniques for detecting attacks in SDN, in: 2019 IEEE 7th International Conference on Computer Science and Network Technology, ICCSNT, 2019, pp. 277–281, http://dx.doi.org/10.1109/ICCSNT47585.2019.8962519.
[19] T.A. Tang, L. Mhamdi, D. McLernon, S.A.R. Zaidi, M. Ghogho, Deep recurrent neural network for intrusion detection in SDN-based networks, in: Proceedings of the 4th IEEE Conference on Network Softwarization and Workshops, NetSoft, Montreal, QC, Canada, 25–29 June 2018, pp. 202–206.
[20] Dey S.K., M. Md Rahman, Flow based anomaly detection in software defined networking: A deep learning approach with feature selection method, in: Proceedings of the 4th International Conference on Electrical Engineering and Information & Communication Technology, iCEEiCT, Dhaka, Bangladesh, 13–15 September 2018, pp. 630–635.
[21] T.A. Tang, L. Mhamdi, D. McLernon, S.A.R. Zaidi, M. Ghogho, Deep learning approach for Network Intrusion Detection in Software Defined Networking, in: Proceedings of the International Conference on Wireless Networks and Mobile Communications, WINCOM, Fez, Morocco, 26–29 October 2016, 2016, pp. 258–263.
[22] K. Malialis, D. Kudenko, Distributed response to network intrusions using multiagent reinforcement learning, Eng. Appl. Artif. Intell. 41 (2015) 270–284.
[23] Yandong Liu, Mianxiong Dong, Kaoru Ota, Jianhua Li, Jun Wu, Deep reinforcement learning based smart mitigation of DDoS flooding in software-defined networks, in: IEEE 23rd International Workshop, CAMAD, 2018.
[24] Y. Feng, J. Li, T. Nguyen, Application-layer DDoS defense with reinforcement learning, in: 2020 IEEE/ACM 28th International Symposium on Quality of Service, IWQoS, 2020, pp. 1–10, http://dx.doi.org/10.1109/IWQoS49365.2020.9213026.
[25] S. Ali, M.K. Alvi, S. Faizullah, M.A. Khan, A. Alshanqiti, I. Khan, Detecting DDoS attack on SDN due to vulnerabilities in OpenFlow, in: 2019 International Conference on Advances in the Emerging Computing Technologies, AECT, 2019, pp. 1–6, http://dx.doi.org/10.1109/AECT47998.2020.9194211.
[26] N. Bizanis, F. Kuipers, Sdn and virtualization solutions for the internet of things: A survey, IEEE Access 4 (2016) 5591–5606.
[27] S. Faizullah, S. AlMutairi, Vulnerabilities in sdn due to separation of data and control planes, Int. J. Comput. Appl. 31 (2018) 21–24.
[28] S. Hameed, H. Khan, Sdn based collaborative scheme for mitigation of ddos attacks, Future Internet 10 (3) (2018) 1–18.
[29] D. Kotani, Y. Okabe, A packet-in message filtering mechanism for protection of control plane in openflow networks, in: Symposium on Architectures for Networking and Communications Systems, 2014, pp. 29–40.
[30] S. Mousavi, M. Hilaire, Early detection of DDoS attacks against sdn controllers, in: International Conference on Computing, Networking and Communications, 2015, pp. 77–81.
[31] M.F. Bari, et al., Dynamic controller provisioning in software defined networks, in: Proceedings of the 9th International Conference on Network and Service Management, CNSM 2013, Zurich, 2013, pp. 18–25.
[32] Hidenobu Aoki, Norihiko Shinomiya, Controller placement problem to enhance performance in multi-domain SDN networks, in: ICN fifth ed., ICN 2016, pp. 108-109, ISBN: 978-1-61208-450-3.
[33] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, R. Kompella, Towards an elastic distributed SDN controller, in Proc. 2nd ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw., 2013, pp. 7–12.
[34] A. Filali, A. Kobbane, M. Elmachkour, S. Cherkaoui, SDN controller assignment and load balancing with minimum quota of processing capacity, in: IEEE International Conference on Communications, ICC'18, 2018, http://dx.doi.org/10.1109/ICC.2018.8422750.
[35] A.El. Kamel, H. Youssef, Improving switch-to-controller assignment with load balancing in multi-controller software defined WAN (SD-WAN), J. Netw. Syst. Manage (2020) http://dx.doi.org/10.1007/s10922-020-09523-2.
[36] M.K. Shin, K.H. Ki-Hyuk Nam, H.J. Kim, Software-defined networking (SDN): A reference architecture and open API, in: Proceedings, 2012 International Conference on ICT Convergence, ICTC, 2012, pp. 360–361.
[37] Software defined networking: A new paradigm for virtual, dynamic, in: Flexible Networking, IBM, 2012.
[38] A. El Kamel, H. Youssef, Improving switch-to-controller assignment with load balancing in multi-controller software defined WAN (SD-WAN), J. Netw. Syst. Manage (2020) http://dx.doi.org/10.1007/s10922-020-09523-2.
[39] Y. Xu, Y. Liu, DDoS attack detection under SDN context, in: Proceedings of the 35th Annual IEEE International Conference on Computer Communications, San Francisco, USA, 2016, pp. 1-9.
[40] A. Alshamrani, A. Chowdhary, S. Pisharody, D. Lu, D. Huang, A defense system for defeating DDoS attacks in sdn based networks, in: International Symposium on Mobility Management and Wireless Access, 2017, pp. 83–92.
[41] L. Yang, H. Zhao, Ddos attack identification and defense using sdn based on machine learning method, in: International Symposium on Pervasive Systems, Algorithms and Networks, 2018, pp. 174–178.
[42] P. Dong, X. Du, H. Zhang, T. Xu, A detection method for a novel ddos attack against sdn controllers by vast new low-traffic flows, in: International Conference on Communications, 2016, pp. 1–6.
[43] N. Dayal, P. Maity, S. Srivastava, R. Khondoker, Research trends in security and ddos in sdn, Secur. Commun. Netw. 9 (18) (2016) 6386–6411.
[44] K. Benton, L.J. Camp, C. Small, Openflow vulnerability assessment, in: Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, New York, USA, 2013, pp. 151-152.
[45] R. Kloti, V. Kotronis, P. Smith, OpenFlow: A security analysis, in: Proceedings of the 8th IEEE ICNP Workshop on Secure Network Protocols, Gottingen, GERMANY, 2013, pp. 1-6.
[46] R. Braga, E. Mota, A. Passito, Lightweight DDos flooding attack detection using nox/openflow, in: Proceedings of the 35th IEEE Conference on Local Computer Networks, Denver, USA, 2010, pp. 408-415.
[47] Fei Wang, Xiaofeng Hu, Jinshu Su, Unfair rate limiting for ddos mitigation based on traffic increasing patterns, in: 2012 IEEE 14th International Conference on Communication Technology, Chengdu, 2012, pp. 733–738, http://dx.doi.org/10.1109/ICCT.2012.6511301.

[48] Heung-Il. Suk, An introduction to neural networks and deep learning, Deep Learn. Med. Image Anal. (2017).
[49] A. Mahmood, et al., Deep learning for coral classification, in: Handbook of Neural Computation, Academic Press, 2017, pp. 383–401, http://dx.doi.org/10.1016/B978-0-12-811318-9.00021-1.
[50] D. Sahoo, Q. Pham, J. Lu, S.C.H. Hoi, Online deep learning: Learning deep neural networks on the fly, in: Proceedings of the 27th International Joint Conference on Artificial Intelligence, IJCAI-18, 2018.
[51] Tore Opsahl, Pietro Panzarasa, Clustering in weighted networks, Social Networks 15 (2009) 5–163, http://dx.doi.org/10.1016/j.socnet.2009.02.002, CiteSeerX 10.1.1.180.9968.
[52] Python 3.0 release, 2020, https://www.python.org/download/releases/3.0/ (Accessed April 2020).
[53] Jupyter lab plateform (R0.33.4), 2020, https://jupyter.org/about.html (Accessed April 2020).
[54] Floodlight controller, 2020, https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview (Accessed 1 october 2020).
[55] Hping3, 2020, http://linux.die.net/man/8/hping3/ (Accessed 3 october 2020).
[56] Mininet overview, 2020, http://mininet.org/overview/ (Accessed 2 october 2020).
[57] Openvswitch 2.5 documentation, 2020, https://www.openvswitch.org/support/dist-docs-2.5/ (Accessed 2 october 2020).