Efficient and Low-cost Defense against Distributed Denial-of-service Attacks in SDN-based Networks

You-Chiun Wang and Yi-Chuan Wang

Abstract—Distributed denial-of-service (DDoS) attacks are common threats in many networks, where attackers attempt to make victim servers unavailable to other users by flooding them with worthless requests. These attacks cannot be easily stopped by firewalls, since they forge lots of connections to victims with various IP addresses. The paper aims to exploit the *software-defined networking (SDN)* technique to defend against DDoS attacks. However, the controller has to handle lots of connections launched by DDoS attacks, which burdens it with a heavy load and degrades SDN's performance. Therefore, the paper proposes an *efficient and low-cost DDoS defense (ELD) mechanism* for SDN. It adopts a nested reverse-exponential data storage scheme to help the controller efficiently record the information of packets in the limited memory. Once there are many packets with high IP variability sent to a certain server, and this situation lasts for a while, then a DDoS attack is likely happening. In this case, the controller asks switches to block malicious connections by installing flow rules. Experimental results verify that the ELD mechanism rapidly recognizes protocol-based DDoS attacks and stops them in time, including TCP SYN flood, UDP flood, and ICMP flood, and also greatly reduces the overhead for the controller to defend against attacks. Moreover, ELD can distinguish DDoS flows from legitimate ones with similar features such as elephant flows and impulse flows, thereby eliminating false alarms.

Index Terms—DDoS attack, elephant flow, impulse flow, IP variability, SDN technique.

1 Introduction

DISTRIBUTED denial-of-service (DDoS) attacks have been a growing problem in the Internet. Attackers target some servers (also known as *victims*) and overwhelm them with a mass of requests, which prevents normal use of their services. This can be easily done by a botnet, which contains compromised devices such as computers, smart phones, or IoT (Internet of Things) devices whose security has been breached [1]. These devices are remotely controlled by attackers to overly ask a victim for connections, resulting in poor performance or even no service available.

In the past, firewalls are usually viewed as the first-line defense against DDoS attacks. A firewall checks every packet routed in and out of a network based on predefined filtering rules like certain IP (Internet protocol) addresses or ports. However, DDoS attacks will flood a firewall with redundant packets, making it busily perform useless work and substantially degrading its performance [2]. Even worse, an attacker can also fabricate IP addresses and ports of packets to make a fool of the firewall.

The emerging *software-defined networking (SDN)* technique facilitates network management by using a controller to centralize control over each switch [3]. Therefore, the controller can manage data transmissions by installing flow rules in switches to tell them how to process packets. With the help of the controller, it becomes much easier for users to monitor the network status and command switches. Thanks to its flexibility, a variety of SDN applications have been proposed, for example, load-balancing control [4], anonymous communications [5], rogue Wi-Fi access point detection [6], and redundant message elimination [7].

In this paper, we investigate how to exploit SDN to fast detect and stop DDoS attacks. Nevertheless, DDoS attacks also affect the performance of SDN. Specifically, lots of connection requests are launched when a DDoS attack occurs. In this case, switches have no idea to process these requests and ask the controller for instructions [8]. Unavoidably, the controller will be busy coping with DDoS flows, resulting in low performance or even paralysis of the SDN network. Therefore, it is a challenge to reduce the workload of the controller (in both computation and rule installation) to defend against DDoS attacks.

Another critical issue is to avoid blocking legitimate connections when defending against DDoS attacks. Certain connections such as impulse flows [4] and elephant flows [9] possess some features similar to DDoS attacks, where they also produce a great deal of packets to be sent to a server, but they are not malicious. When these connections have the same destination with a DDoS attack, a switch may treat their packets to be parts of the attack, thereby dropping the packets and blocking them accordingly. This is a side effect of DDoS defense (i.e., false alarms) but does attain the aim of DDoS attacks (i.e., preventing legitimate use of the victim's service).

In view of the above motivations, this paper develops an *efficient and low-cost DDoS defense (ELD)* mechanism for SDN with three objectives: 1) quickly recognize DDoS attacks, 2) reduce the controller's overhead, and 3) avoid false alarms. We aim at protocol-based DDoS attacks, including TCP (transmission control protocol) SYN (synchronization) flood, UDP (user datagram protocol) flood, and ICMP (Internet control message protocol) flood. To facilitate identification of DDoS attacks, the controller refers to three signatures of traffic flows, including flow size, IP variability, and duration. To do so, it needs to keep recording the related information of packets. Since the memory space is limited, we also propose a *nested reverse-exponential storage (NRES)* scheme in ELD to help the controller record more recent packets and also keep a few old packets for reference.

The contributions of this paper are threefold. First, unlike

The authors are with the Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung, 80424, Taiwan. E-mail: ycwang@cse.nsysu.edu.tw; m053040075@student.nsysu.edu.tw

some SDN-based approaches that look over many signatures of packets (which burdens the controller with a heavy load), we propose a lightweight defense mechanism against DDoS attacks by checking only flow size, IP variability, and duration. It thus can greatly reduce the computational cost of the controller. Second, we develop the NRES scheme to help the controller efficiently store the necessary information of packets in its limited memory. NRES records more recent packets while keeping some ancient packets for a long history, so it can provide a broad and high-resolution view of attacks. Third, our ELD mechanism can differentiate between DDoS flows and legitimate flows that share similar features (in particular, elephant flows and impulse flows). Therefore, the packets of these legitimate flows will not be erroneously discarded. Through simulations by Mininet, we validate that ELD blocks more DDoS packets, eliminates false alarms, and significantly lowers the controller's overhead on computation and rule installation, as compared with other SDN-based approaches.

This paper is organized as follows: Section 2 gives background knowledge and Section 3 surveys related work. We discuss the system model in Section 4, propose the ELD mechanism in Section 5, and present performance evaluation in Section 6. Finally, a conclusion is drawn in Section 7.

2 Background Knowledge

2.1 Protocol-based DDoS Attacks

There are three typical types of protocol-based DDoS attacks. A *TCP SYN flood attack* spoils the three-way handshake mechanism in TCP [10]. When two hosts want to set up a TCP connection, the source first sends an SYN packet and the destination then replies an SYN-ACK (acknowledgement) packet. Afterward, the source sends an ACK packet to finish the handshake procedure. However, an attacker can send lots of SYN packets to a victim with spoofed IP addresses. In this case, the victim replies many SYN-ACK packets and uses a TCP port for each connection. Expectably, there will be no ACK packets sent from these sources, as they may be fabricated or those compromised devices that are not aware of generating the attack. However, the victim has to keep its TCP ports open until timeout. Since the victim is busy replying SYN-ACK packets and may use up its TCP ports, its service will be eventually disrupted.

In a *UDP flood attack*, numerous UDP requests (possibly with spoofed IP addresses) are forwarded to a victim [11]. The UDP ports of these requests could be random or dedicated. When the victim gets a UDP request, it checks whether the corresponding UDP port is open (i.e., the victim provides the UDP service). If not, the victim replies an ICMP packet to notify the source. Inevitably, the victim has to do lots of computation to handle UDP requests, and its link to the network will be congested with many ICMP packets.

An *ICMP flood attack* is usually accomplished by the ping service [12]. The attacker transmits lots of ping requests to other devices in the network whose source addresses are the victim's IP address. In this case, these devices return a great deal of ping replies to the victim and exhaust its bandwidth accordingly. Alternatively, the attacker keeps sending many ping requests with spoofed IP addresses to the victim. Thus, the victim will be fully occupied answering ping replies. Such attacks can be also realized by other ICMP services such as echo.

2.2 Elephant Flows and Impulse Flows

Generally speaking, *elephant flows* carry volumes of data and last for a long time. They are usually produced for bulk data transfer, for example, migration of virtual machines or system backup [9]. Elephant flows are popular in some networks such as data center networks. In fact, Lin et al. [13] point out that nearly 90% of data bytes in these networks are contributed by elephant flows, but they account for just 1% of total flows. Thus, the sources of elephant flows are concentrated (i.e., there are fewer sources to generate such flows). On the other hand, *impulse flows* also possess many data but last for a short time [4]. They could be generated when many users provisionally query resources from servers, such as watching video chips or downloading files. Unlike elephant flows, the sources of impulse flows may be more diverse.

Both elephant flows and impulse flows will produce lots of packets (in different time spans) and substantially consume a server's bandwidth, making them behave similarly with DDoS attacks [14]. However, they are completely legitimate and normal flows. Consequently, we should distinguish these flows from DDoS ones to avoid blocking them when stopping DDoS attacks.

2.3 SDN and OpenFlow

The traditional network architecture makes both control and data planes be tightly coupled in every switch. Therefore, switches decide how to process and route packets on their own. Nevertheless, this design complicates network management, because administrators have to configure switches one by one when new policies are applied to the network. SDN simplifies network management by decoupling the two planes, where the control plane is put in a central *controller*, whereas the data plane is distributed among switches. In this way, the whole network is in the grasp of the controller, and administrators can write programs in the controller to easily configure and command switches [15].

The OpenFlow protocol is a popular SDN implementation, which is regulated by the open networking foundation (ONF). OpenFlow provides an application program interface for communications between the controller and switches. In particular, each switch maintains *flow tables* composed of flow entries. A *flow entry* has match fields to tell the switch whether a packet meets certain conditions. If so, the switch follows the entry's instructions to process that packet. The controller can set up a TLS (transport layer security) connection with each switch, and install new entries in its flow table or remove old ones. Thus, the switch's behavior will be changed accordingly.

3 Related Work

There have been a number of approaches proposed to defend against DDoS attacks. Below, we discuss the approaches used in traditional networks, followed by SDN-based approaches.

3.1 Traditional Approaches

Intrusion detection systems are widely used to discover anomalous activities and policy violations in a network [16]. They can be placed on the entrance to a subnet where firewalls are located to analyze inbound and outbound traffics, so as to check whether the behavior of some flows match with known attacks. For example, Pengfule et al. [17] regularly measure the number of SYN packets and the statistics of flows. If there are too many SYN packets (as compared with total flows), it is viewed as a TCP SYN flood attack, so subsequent SYN packets are blocked by the firewall. However, this approach incurs a high false-alarm ratio when there exist elephant flows, because they occupy most of the network bandwidth, making the ratio of SYN packets to total flows exceed the threshold.

IP traceback is one popular mechanism to find out the sources of DDoS and IP spoofing attacks [18]. There are three common approaches to carry out IP traceback: *link testing*, *logging*, and *ICMP traceback*. Beginning from the victim, the link testing approach [19] traces packets link by link upstream, which can eventually reach the source of an attack. Nevertheless, it involves the cooperation of multiple Internet service providers to support link testing. The logging approach [20] asks each router to store information of passing packets (e.g., IP addresses and packet counts). In case of attacks, the victim can use this information for investigation. However, routers spend much memory space to store packet information and also large computational power to process packets for logging. In the ICMP traceback approach [21], each router sends ICMP packets to its previous-hop and next-hop routers with a probability to verify IP addresses of passing packets. Thus, it helps the victim identify the path of a DDoS attack. However, more ICMP packets will be produced when the path becomes longer, which consumes more bandwidth and congests the network.

To overcome TCP SYN flood attacks, one past solution is to shorten the timeout of SYN packets [22]. It enables the victim to fast discard those semi-connections generated by an attack. Another solution is to associate each connection with one SYN cookie [23]. If the victim gets multiple SYN packets from the same source in a short time, they are treated as parts of an attack and dropped accordingly. However, these two solutions will fail when the attacker sends lots of SYN packets with different IP addresses (e.g., generating random IP addresses by SOCK_RAW).

Checking IP validness of packets also helps defend against UDP flood attacks. One representative is *unicast reverse-path forwarding* [24]. When a router receives one packet, it checks whether there exists a reverse path back to the source by consulting the routing table. If not, the packet may be forged by an attacker, and the packet will be dropped accordingly. By considering the discontinuity of IP addresses, Xu et al. [25] propose a negative selection algorithm based on the concept of eigenvalue set to find out fake IP addresses. However, the above approaches rely on routers to verify every packet, which inevitably burdens them with heavy loads.

3.2 SDN-based Approaches

Rengaraju et al. [26] implement an *intrusion prevention system (IPS)* on software-defined clouds (e.g., IBM SmartCloud [27]) to conquer TCP SYN and ICMP flood attacks. Each switch uses the IPS module to monitor traffic flows and reports to the controller. Then, the firewall installed in the controller checks the signatures of every flow, including flow size, packet length, and TCP flags, for recognizing attacks. However, large-sized elephant flows are not considered, which may increase the incidence of false alarms. Besides, some signatures like packet length can be easily falsified by an attacker, resulting in misjudgement.

SLICOTS [28] is a lightweight countermeasure for TCP SYN flood attacks based on SDN. Once detecting potential attacks, the controller adds a flow entry to each corresponding switch to drop malicious SYN packets, whose Short_Hard_TimeOut field is set to T seconds (e.g., $T = 3$). In other words, this flow entry will be removed by the switch after T seconds. The advantage of SLICOTS is that the controller need not take care of the removal of flow entries. However, if the duration of an attack is much longer than T seconds, the controller has to repeatedly detect the attack and install the same flow entries, which wastes its computational resources.

For UDP flood attacks, one feature is that the number of UDP responses will be much larger than that of UDP requests [29]. Based on this observation, Mutu et al. [30] use the SDN controller to measure the ratio of UDP packets sent to a target server to total packets. If the ratio exceeds a threshold, an attack may be happening. However, how to block the packets generated by attacks to stop them is not addressed. In the work [31], the controller periodically queries each switch for the number of UDP packets passing through its ports. If a port has more received packets than transmitted packets, it infers that the destinations of some transmitted packets do not exist in the network. In this case, the controller asks the switch discarding these packets. However, an elephant flow usually has many received packets but just few transmitted packets. Such normal flows will be also blocked by this approach.

Yu et al. [32] develop an SDN-based DDoS detection and response platform for vehicular networks. The proposed platform estimates the frequency of sending Packet_In messages to the controller. Once the frequency overtakes the predefined threshold, a feature extraction strategy is used to analyze the entropy of flow tables, IP addresses, and flow counts to determine whether a DDoS attack is happening. Nevertheless, they do not consider the impact of elephant flows and impulse flows. In addition, how to efficiently store the information of flows for feature extraction is not discussed.

In SDNScore [33], each switch adopts eight tables to record packet signatures: source IP address, destination IP address, source port, destination port, protocol type, packet size, time-to-live value, and TCP flags. To establish the relationship between any two signatures, the controller maintains $\hat{C}(8,2) = 28$ tables, where $\hat{C}(\cdot,\cdot)$ denotes the function of combinations. When attacks occur, the controller grades each packet (by the 28 tables) to decide whether to send or drop the packet. Although SDNScore provides a sophisticated defense method against DDoS attacks, the controller and switches have to spend much memory space to keep the above tables. Besides, the controller will be unavoidably busy grading packets during the occurrence of an attack.

As compared with the prior work, our paper aims to provide a low-cost defense solution against DDoS attacks with fewer false alarms when there exist elephant flows and impulse flows in the network. Furthermore, we also propose an efficient data storage scheme to help the controller store necessary packet information in its limited memory. These designs distinguish our paper from the prior work.

4 System Model

Fig. 1 presents a schematic diagram for DDoS attacks and their defenses by SDN. Specifically, let us consider a *target network* composed of hosts and OpenFlow switches. A controller takes charge of managing the target network. Each OpenFlow switch notifies the controller of its status (e.g., the amount of data passed through its ports) by using Packet_In messages. On

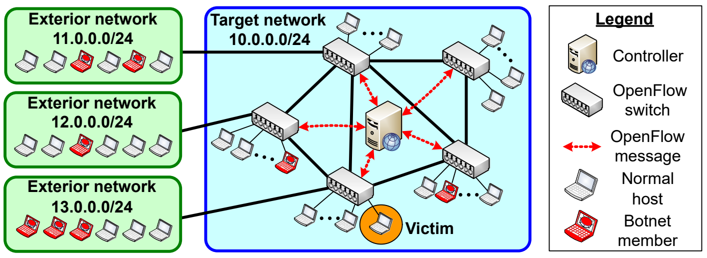

Exterior network 11.0.0.0/24
Exterior network 12.0.0.0/24
Exterior network 13.0.0.0/24
Target network 10.0.0.0/24
Victim
Legend
Controller
OpenFlow switch
OpenFlow message
Normal host
Botnet member

Fig. 1: Schematic diagram for DDoS attacks and their defenses by SDN.

TABLE 1: Summary of notations.

Notations	Definitions
Φ	The set of all ports of the switches in the target network
f_{avg}	Average flow size
r_i	The record to store information of a packet p_i
n	The number of records that can be stored in the memory
t	The current time
$\mathcal{R}_x$	The set of records for a port $\varphi_x \in \Phi$
$\mathcal{G}_i$	Group of records in $\mathcal{R}_x$ whose source IP addresses are a_i
V_x	IP variability of the records in $\mathcal{R}_x$
δ	Threshold on V_x to check if the IP variability is high
c	A counter used in the impulse flow checking module
β	The initial value of counter c

the other hand, the controller can give instructions to an OpenFlow switch by sending it a Packet_Out message that contains flow entries to be installed (or removed).

An attacker will pick some hosts in the target network as victims and manipulate a botnet to send lots of DDoS packets to the selected victims at any time. Each member of the botnet is a host that has been compromised by the attacker, possibly through computer viruses or Trojan horses. These botnet members may reside in exterior networks or could be even hosts in the target network, as illustrated in Fig. 1.

Some interior hosts (i.e., in the target network) and exterior hosts (i.e., outside the target network) would produce normal and legitimate flows whose destinations are also the victims, including ordinary flows, elephant flows, and impulse flows. Since the members of the botnet may be unknown and could dynamically change, our problem asks how to quickly detect DDoS attacks and stop them, such that not only the controller's overhead (in computation and SDN messages) is reduced but also false alarms (caused by elephant flows and impulse flows) can be minimized. Table 1 summarizes the notations used in the paper.

5 The Proposed ELD Mechanism

As discussed earlier, a DDoS attack is usually accompanied by lots of packets with various source IP addresses. Moreover, it would last for a relatively long time in order to disrupt the victim's service. Consequently, three principles are adopted in the ELD mechanism for the identification of DDoS attacks:

- **Principle 1:** Is the amount of traffics (i.e., flow size) too large?
- **Principle 2:** Is the variability of source IP addresses too high?
- **Principle 3:** Is the duration of the above situation too long?

Generally speaking, principle 1 is applied to check whether potential network congestion occurs, which may be a prelude of DDoS attacks. On the other hand, principles 2 and 3 are used to distinguish between elephant/impulse flows and DDoS ones. When the checking conditions of these three principles are all satisfied, there is a high possibility that an attack is happening. In this case, the controller should take actions to stop the attack by installing flow rules in OpenFlow switches. Obviously, the controller needs to record the information of packets to make decisions on whether DDoS attacks occur by the above principles. However, since it has limited memory space, we thus develop a data storage scheme to help the controller efficiently perform its task.

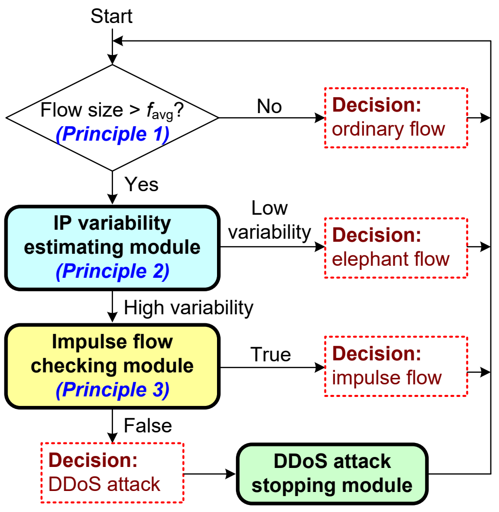


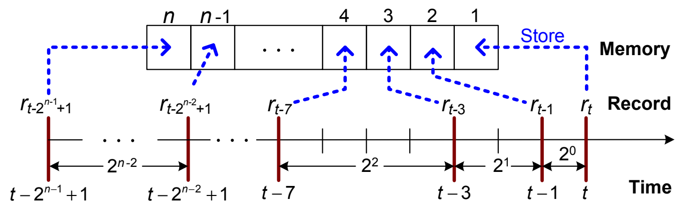
Fig. 2: Flowchart of the ELD mechanism.

Algorithm 1 The ELD mechanism

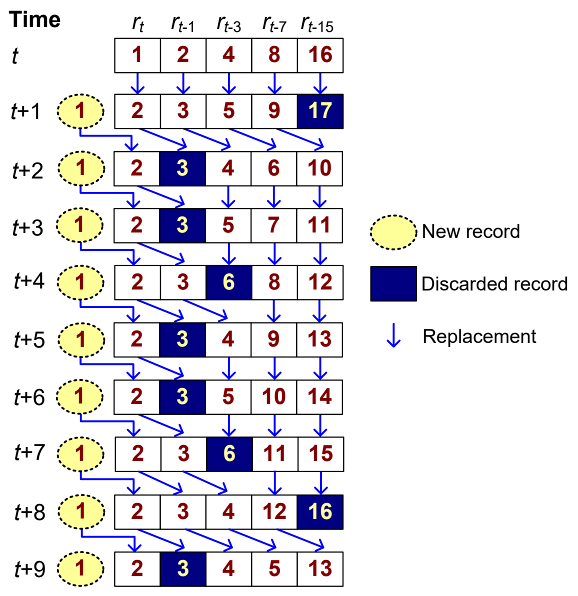
```
1: for each port $\varphi_x \in \Phi$ do
2:     Let $f_x$ be the flow size of $\varphi_x$;
3:     if $f_x > f_{\text{avg}}$ then
4:         if IVE($\varphi_x$) = high then
5:             if IFC($\varphi_x$) = false then
6:                 Call DAS($\varphi_x$) to stop attacks;
7:             end if
8:         end if
9:     end if
10: end for
```

Fig. 2 presents the flowchart of the ELD mechanism. Specifically, each switch keeps monitoring the amount of traffics sent through every port, which is known as the *flow size*. When the flow size of a port goes beyond the average value f_{avg} in the network (i.e., principle 1), the switch notifies the controller of this incident. After that, the controller invokes the *IP variability estimating module* to assess the variability of source IP addresses for that port (i.e., principle 2). If the IP variability is low, the controller judges that this is a safe case of elephant flows. Otherwise, the controller further uses the *impulse flow checking module* to determine whether this is a safe case of impulse flows (i.e, principle 3). Once the module's check cannot be passed (i.e., returning false), the controller infers that a DDoS attack is happening. In this case, the controller executes the *DDoS attack stopping module* to ask the switch dropping malicious packets, and then recovers the switch's setting after the attack terminates.

Based on the flowchart in Fig. 2, we present the pseudocode of our ELD mechanism in Algorithm 1. Let Φ denote the set of all ports of the switches in the target network. The



(a) Ideal case



(b) Practical example

Fig. 3: The original RES scheme.

controller keeps monitoring each port in Φ and examines if any port φ_x meets the checking conditions of the three principles. Specifically, the code in line 3 shows the checking condition of principle 1. The checking condition of principle 2 is indicated in line 4, where the IVE(φ_x) procedure in Algorithm 2 will conduct the IP variability estimating module. Then, the checking condition of principle 3 is given in line 5, where the IFC(φ_x) procedure in Algorithm 3 will perform the impulse flow checking module. Once the three if-conditions in lines 3–5 are all satisfied, a DDoS attack is likely happening. Thus, the DAS(φ_x) procedure in Algorithm 4 is called to carry out the DDoS attack stopping module.

Below, we first propose our data storage scheme, then elaborate on the design of each module, and finally make a discussion on the ELD mechanism.

5.1 Data Storage Scheme

The controller takes down IP data for each port $\varphi_x \in \Phi$. Specifically, for every packet p_i sent via port φ_x, a three-tuple record $r_i = (t_i, a_i, w_i)$ is produced, where t_i is the time when p_i was sent, a_i is the source IP address of p_i, and w_i is the IP protocol number. Each switch can produce records and keep them in the local memory, and periodically send a batch of records to the controller for the sake of efficiency. To find out DDoS attacks, the controller has to store a long history of records as reference. Besides, more records of recent packets should be also available. Since the memory space of the controller is limited, these two objectives may conflict with each other. To conquer the dilemma, we borrow the notion of the *reverse-exponential storage (RES)* scheme [34].

Suppose that at most n records can be kept in the memory and the current time is t. RES seeks to store historical records at timestamps with intervals in an exponentially incremental order from t. In an ideal case, RES stores records r_t, r_{t-1}, r_{t-3}, $\cdots$, and $r_{t-2^{n-1}+1}$, as shown in Fig. 3(a). Thus, the controller has records long time ago with different resolutions. In practical implementation, when time moves to $t+1$, the site for r_t is given to r_{t+1}, and the site for r_{t-1} is given to r_t. Then, the site for each $r_{t-2^\tau+1}$, $2 \leq \tau \leq n-1$, is given to the record whose original timestamp is closest to $t-2^\tau+1$. An exception is $r_{t-2^{n-1}+1}$, where the records with timestamps older than $t-2^{n-1}+1$ are discarded. Fig. 3(b) gives an example, where each number indicates the real age of a record at each time instance. Suppose that the memory has historical records with ages 1, 2, 4, 8, and 16 in sites r_t, r_{t-1}, r_{t-3}, r_{t-7}, and r_{t-15} at time t, respectively. When time moves to $t+1$, the age of each stored record is added by one. Thus, the records with ages 1, 2, 3, 5, and 9 are kept. However, the record with age 17 is discarded because it is too old. Fig. 3(b) shows the results from time t to $t+9$. Theorem 1 then discusses the property of RES.

Theorem 1. Given a memory with n records, the timestamp gap between an actual record and its ideal case is no larger than 2^{n-2} in RES, where $n \geq 2$.

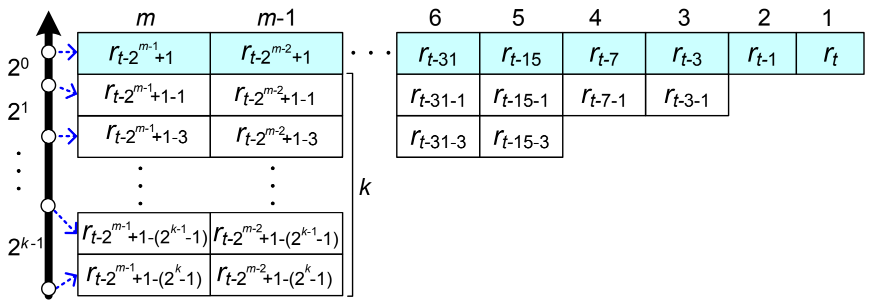
Proof: Observing from Fig. 3(b), the sites for r_t and r_{t-1} are given to records r_{t+1} and r_t when time moves to $t+1$, respectively, so the records kept in both sites r_t and r_{t-1} must be correct. For each of other sites $r_{t-2^\tau+1}$, $\tau = 2, \cdots, n-1$, the actual record r_α stored in that site will meet the condition of $|\alpha - (t-2^\tau+1)| \leq 2^{\tau-1}$, because the original record stored in site $r_{t-2^{\tau-1}+1}$ will eventually move to site $r_{t-2^\tau+1}$. Since $\tau \leq n-1$, the maximum timestamp gap is thus $2^{(n-1)-1} = 2^{n-2}$. □

However, RES was originally designed for those devices with very little memory space (e.g., small sensors). When n becomes large, there will be more outdated records kept in the memory. These records are too old as compared with the lifespan of a DDoS attack. Thus, RES cannot be directly applied to data storage for the controller, since many stored records are pretty old and actually useless.

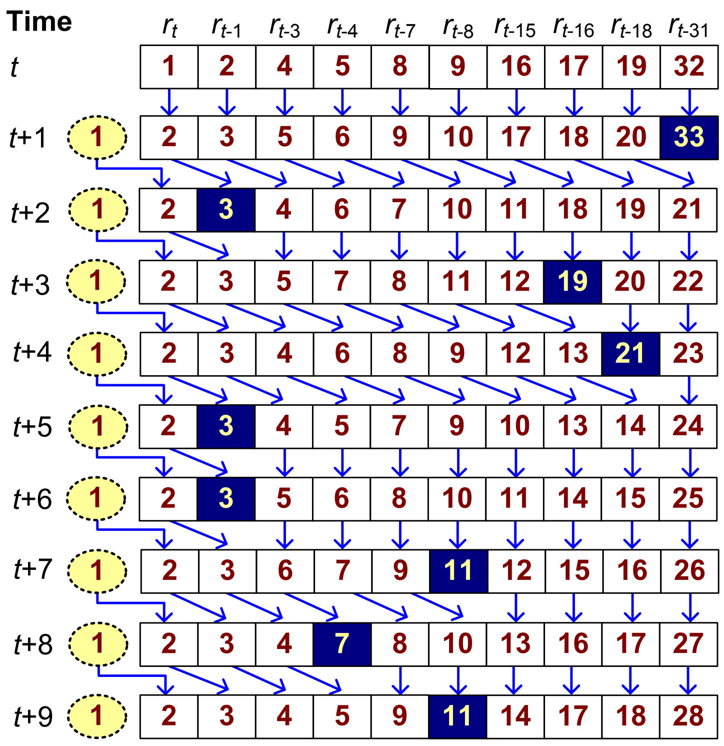
To solve this problem, we tailor RES to practical needs by a *nested RES (NRES)* idea, as illustrated in Fig. 4(a). Similar to RES, NRES also stores records r_t, r_{t-1}, r_{t-3}, $\cdots$, and $r_{t-2^{m-1}+1}$, where $m < n$. For ease of presentation, let us set $r_t, r_{t-1}, r_{t-3}, \cdots$, and $r_{t-2^{m-1}+1}$ as the first records of *columns* 1, 2, 3, $\cdots$, and m, respectively. For each pair of columns τ and $\tau+1$, they can store additional $(\tau-1)/2$ records. In particular, if m is odd, we have $\tau = 1, 3, \cdots, m-2$ and column m can also store extra $(m-1)/2$ records. Otherwise, we have $\tau = 1, 3, \cdots, m-1$. Then, we follow the rule of RES to store records in each column. Specifically, records $r_{t-2^{\tau-1}+1}, r_{t-2^{\tau-1}+1-1}, r_{t-2^{\tau-1}+1-3}, \cdots, r_{t-2^{\tau-1}+1-(2^{k-1}-1)}$, and $r_{t-2^{\tau-1}+1-(2^k-1)}$ will be stored in column τ, where $k = \left\lfloor \frac{\tau-1}{2} \right\rfloor$. Fig. 4(a) presents an ideal case of our proposed NRES scheme.

The remaining issue is how to decide the value of m, which depends on the size n (in records) of the memory. In particular, we find the smallest *odd* value of m that satisfies the following inequality:

$$\frac{m-1}{2} \times \left\lfloor \frac{m-2}{2} \right\rfloor + \frac{3m-1}{2} \geq n. \tag{1}$$



(a) Ideal case, where $k = \lfloor \frac{m-1}{2} \rfloor$



(b) Practical example

Fig. 4: Our proposed NRES scheme.

Let m_o be the answer in Eq. (1). Afterward, we also find the smallest *even* value of m to satisfy the following inequality:

$$\frac{m}{2} \times \left\lfloor \frac{m-1}{2} \right\rfloor + m \geq n. \tag{2}$$

Similarly, let m_e be the answer in Eq. (2). Then, the final value of m is determined by

$$m = \min\{m_o, m_e\}. \tag{3}$$

Theorem 2 proves the correctness of our selection for the value of m. Next, Theorem 3 gives an analysis on the timestamp of the oldest record stored by the NRES scheme.

Theorem 2. Suppose that at most n records can be kept in the memory. Then, Eq. (3) must hold in the NRES scheme.

Proof: Observing from Fig. 4(a), NRES adds one more record for each pair of columns in sequence. Therefore, the number of records in each pair of columns will form an arithmetic progression. Suppose that m is odd. Then, the maximum number of records stored in the memory can be calculated by

$$\left(2 \times \frac{\frac{m-1}{2} \times \left(1 + \lfloor \frac{m-2}{2} \rfloor + 1\right)}{2}\right) + \left(\frac{m-1}{2} + 1\right) \geq n. \tag{4}$$

In Eq. (4), the first term gives the total number of records stored in columns 1, 2, $\cdots$, and $m-1$, and the second term indicates the maximum number of records stored in column m. Since the oldest record must locate in column m (but may not necessarily be the last record of this column in the ideal case), the sum of these two terms will be no smaller than n. By doing some algebraic operations, Eq. (4) can be simplified to Eq. (1). On the other hand, if m is even, the maximum number of records stored in the memory will be computed by

$$2 \times \frac{\frac{m}{2} \times \left(1 + \lfloor \frac{m-1}{2} \rfloor + 1\right)}{2} \geq n. \tag{5}$$

Similarly, Eq. (5) can be simplified to Eq. (2). Evidently, the value of m should be the minimum answer that satisfies both Eqs. (1) and (2), which verifies this theorem. □

Theorem 3. Let t be the current time. Then, the timestamp of the oldest record kept in the memory by NRES is $t - 2^{m-1} + 1 - \left(2^{\varepsilon-1} - 1\right)$, where $\varepsilon = n - \frac{m-1}{2} \times \left(\lfloor \frac{m-2}{2} \rfloor + 2\right)$ if m is odd, or $\varepsilon = n - \frac{m-2}{2} \times \left(\lfloor \frac{m-3}{2} \rfloor + 2\rfloor\right) - \left(\lfloor \frac{m-1}{2} \rfloor + 1\rfloor\right)$ otherwise.

Proof: Due to the storage structure of NRES, the oldest record must reside in column m, which implies that each of columns 1 to $m-1$ can fully store its records. If m is odd, the number of records stored in the first $(m-1)$ columns is $\frac{m-1}{2} \times (\lfloor\frac{m-2}{2}\rfloor + 2)$, so there remains $\varepsilon = n - \frac{m-1}{2} \times (\lfloor\frac{m-2}{2}\rfloor + 2)$ records in column m. On the other hand, when m is even, the number of records stored in the first $(m-2)$ columns is $\frac{m-2}{2} \times (\lfloor\frac{m-3}{2}\rfloor + 2)$. Besides, the number of records stored in column $m-1$ is $(\lfloor\frac{m-1}{2}\rfloor + 1\rfloor)$. Therefore, there are $\varepsilon = n - \frac{m-2}{2} \times (\lfloor\frac{m-3}{2}\rfloor + 2\rfloor) - (\lfloor\frac{m-1}{2}\rfloor + 1\rfloor)$ records left in column m. Observing from Fig. 4(a), the ε-th record stored in column m has timestamp of $t - 2^{m-1} + 1 - (2^{\varepsilon-1} - 1)$, so the theorem is proven. □

Fig. 4(b) gives an example, where $n = 10$. By Eqs. (1) and (2), we have $m_o = 7$ and $m_e = 6$, so m is 6. According to Theorem 3, the timestamp of the oldest record will be $t - 2^{6-1} + 1 - (2^{\varepsilon-1} - 1)$, where $\varepsilon = 10 - \frac{6-2}{2} \times (\lfloor\frac{6-3}{2}\rfloor + 2\rfloor) - (\lfloor\frac{6-1}{2}\rfloor + 1\rfloor) = 1$. Thus, the timestamp is $t - 31$. Suppose that the memory has historical records with ages 1, 2, 4, 5, 8, 9, 16, 17, 19, and 32 in sites r_t, r_{t-1}, r_{t-3}, r_{t-4}, r_{t-7}, r_{t-8}, r_{t-15}, r_{t-16}, r_{t-18}, and r_{t-31} at time t, respectively. When time moves to $t+1$, the age of each stored record is added by one. Thus, the records with ages 1, 2, 3, 5, 6, 9, 10, 17, 18, and 20 are kept. The record with age 33 is discarded since it is too old. Fig. 4(b) shows the results from t to $t+9$. We remark that the original RES scheme will keep records with ages 1, 2, 4, 8, 16, 32, 64, 128, 256, and 1024, where many outdated records are left in the memory. Compared with RES, our NRES scheme can store much more recent (and informative) records, which better utilizes the controller's memory space.

5.2 IP Variability Estimating Module

When a DDoS attack is under way, there would be a large number of packets with different source IP addresses flooded into a victim. Therefore, high IP variability of packets can be considered a symptom of attacks (i.e., principle 2).

Let $\mathcal{R}_x$ be the set of records kept in the controller's memory for a port $\varphi_x \in \Phi$. Besides, let $\mathcal{G}_i$ denote the group of records in $\mathcal{R}_x$ whose source IP addresses are a_i. The condition of $\mathcal{G}_i \cap \mathcal{G}_j = \emptyset$ must hold for $a_i \neq a_j$, where the notation '$\cap$' denotes the operator of intersection, because each record has only one source IP address. On the other hand, we have $\bigcup_{\forall a_i} \mathcal{G}_i = \mathcal{R}_x$, where the notation '$\bigcup$' denotes the operator of union, which means that all $\mathcal{G}_i$ groups comprise the set $\mathcal{R}_x$. Assume that all records in $\mathcal{R}_x$ are divided into N_G groups. The IP variability for port φ_x is then defined by

$$V_x = \frac{N_G}{|\mathcal{R}_x|} \times 100\%, \tag{6}$$

where $|\mathcal{R}_x|$ is the number of records in $\mathcal{R}_x$. For example, suppose that $\mathcal{R}_x$ has five records whose source IP addresses are 10.0.0.1, 10.0.0.2, 10.0.0.2, 10.0.0.3, and 10.0.0.3. Obviously, all records in $\mathcal{R}_x$ can be divided into three groups, so the IP variability V_x is thus $\frac{3}{5} \times 100\% = 60\%$.

From Eq. (6), a larger V_x value implicitly implies that each group $\mathcal{G}_i$ contains fewer records. According to Theorem 3, the NRES scheme lets the controller keep records of packets sent within a fixed period (i.e., from time $t - 2^{m-1} + 1 - (2^{\varepsilon-1} - 1)$ to time t). In other words, the host connecting to port φ_x receives packets from more sources, where each source sends just few packets. In this case, there is a high possibility of attacks to the host. Based on this observation, the controller adopts a threshold δ to judge whether the IP variability is too high. In case of $V_x > \delta$, the controller further checks whether this

Algorithm 2 IP Variability Estimating (IVE) Module

```
1:  procedure IVE($\varphi_x$)
2:      for each record $r_i \in \mathcal{R}_x$ do
3:          if the source IP address of $r_i$ is $a_i$ then
4:              add $r_i$ to group $\mathcal{G}_i$;
5:          end if
6:      end for
7:      $N_G \leftarrow 0$;
8:      for each group $\mathcal{G}_i \subseteq \mathcal{R}_x$ do
9:          $N_G \leftarrow N_G + 1$;
10:     end for
11:     $V_x \leftarrow \frac{N_G}{|\mathcal{R}_x|} \times 100\%$;
12:     if $V_x > \delta$ then
13:         return high;
14:     else
15:         return low;
16:     end if
17: end procedure
```

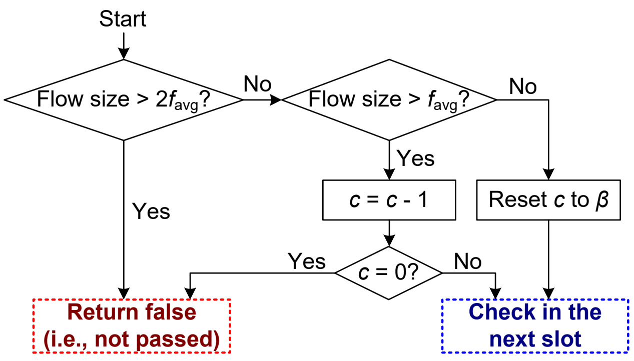


Fig. 5: The flowchart to check impulse flows in a slot.

incident (i.e., high IP variability) is caused by impulse flows or DDoS attacks by the impulse flow checking module in Section 5.3. Otherwise, the controller infers that the packets of elephant flows are currently sent to the host (i.e., a safe case), as shown in Fig. 2.

Algorithm 2 gives the pseudocode of the IP variability estimating module, where the only parameter is the port to be checked (i.e., φ_x). In lines 2–6, we divide the records in $\mathcal{R}_x$ into groups based on their source IP addresses. Then, the code in lines 7–10 calculates the number N_G of groups in $\mathcal{R}_x$. By finding the IP variability V_x for port φ_x in line 11, the module can return either *high* or *low* depending on whether $V_x > \delta$ or not, respectively, as shown in lines 12–16.

It is conceivable that the threshold δ decides the accuracy of attack recognition. In particular, a too large δ value impedes the controller to rapidly recognize DDoS attacks, forcing the victim to keep receiving lots of malicious packets. On the contrary, a too small δ value leads the controller to treat the packets of some normal connections (e.g., elephant flows) as parts of an attack, thereby causing many false alarms and erroneously blocking their packets. In Section 6.3, we will make a further discussion on the effect of δ, where it is suggested to set δ between 80% and 90%.

5.3 Impulse Flow Checking Module

As discussed in Section 2.2, impulse flows usually possess three signatures: 1) carrying large amount of data, 2) originating from many sources, and 3) lasting for a relatively

Algorithm 3 Impulse Flow Checking (IFC) Module

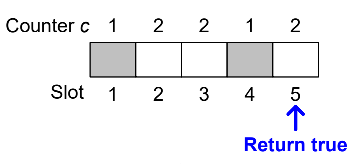
```
1: procedure IFC($\varphi_x$)
2:     Let $f_{x,y}$ be the flow size for port $\varphi_x$ at slot $y$;
3:     $y \leftarrow 1$; $c \leftarrow \beta$;
4:     while $y \le T_{\text{obs}}$ do
5:         if $f_{x,y} > 2f_{\text{avg}}$ then
6:             return false;
7:         end if
8:         if $f_{x,y} > f_{\text{avg}}$ then
9:             $c \leftarrow c - 1$;
10:            if $c = 0$ then
11:                return false;
12:            end if
13:        else
14:            $c \leftarrow \beta$;
15:        end if
16:        $y \leftarrow y + 1$;
17:    end while
18:    return true;
19: end procedure
```

short time. Therefore, if the flow size is large (i.e., overtaking f_{avg}) and the IP variability is also high, the controller should determine whether this flow is impulse or DDoS (i.e., principle 3). To do so, the controller monitors the flow size for a number T_{obs} of slots, namely the *observing period*. If the flow size keeps larger than f_{avg} for β consecutive slots, the controller infers that an attack is happening. Otherwise, the check by this module is passed after the observing period ends, and the controller infers that this incident is caused by impulse flows (i.e., a safe case).

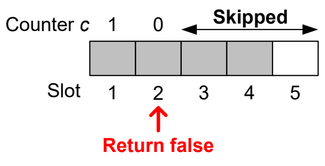
Fig. 5 shows the flowchart to check impulse flows in each slot. Specifically, the controller uses a counter c to do the check, whose initial value is β. In case that the flow size exceeds $2f_{\text{avg}}$, which implicitly implies that the victim is being overwhelmed with lots of packets (whose IP variability is large), there is a high possibility that an attack is launching. Thus, the module returns *false* (i.e., not passed), and the controller need not do the check in the next slot. Otherwise, the controller decreases counter c by one if the flow size overtakes f_{avg}, or resets counter c to the default value (i.e., β). When counter c is either positive or reset to β, the controller does the check in the next slot. Otherwise (i.e., counter c reaches to zero), the module returns false. The above procedure is repeated, until the observing period finishes. In this case, the module returns *true* (i.e., passed).

Algorithm 3 presents the pseudocode of the impulse flow checking module, where the counter c is initially set to β in line 3. The code in lines 5–7 corresponds to the left part of the flowchart in Fig. 5, while the code in lines 8–15 implements the right part of the flowchart. The while-loop will be repeated until y exceeds T_{obs} (i.e., we have checked all slots in the observing period). In this case, the impulse flow checking module will return true in line 18, which means that the check is passed.

Fig. 6 gives two examples, where β is set to 2. A gray slot means that the flow size is between f_{avg} and $2f_{\text{avg}}$, while a white slot means that the flow size is below f_{avg}. In Fig. 6(a), since slots 1 and 4 are not consecutive, they are viewed as impulse flows and the module will return true after slot 5. On the other hand, counter c reaches zero at slot 2 in Fig. 6(b), so



(a) Case of "passed"



(b) Case of "not passed"

Fig. 6: Two examples to illustrate how the impulse flow checking module works.

TABLE 2: The meaning of each bit in the "tcp_flags" field.

Bits	Meanings
0–5	Original TCP flags defined in RFC 793 [35]
6–8	Extra TCP flags defined in RFC 3168 and 3540 [36], [37]
9–11	Reserved bits not yet standardized by IETF
12–15	Set to zero

the module directly returns false at slot 2, and residual slots are skipped accordingly.

It is worth noting that a *short-term and low-volume (STLV)* DDoS attack whose duration is shorter than β slots and flow size is below $2f_{\text{avg}}$ will be passed by the module's check. However, STLV attacks have insignificant impact on network performance, because they produce not many packets and soon disappear. Thus, we do not distinguish them from impulse flows, since doing so will raise the controller's computational cost with just little benefit. Instead, the controller can record the detection of impulse flows (possibly with STLV attacks) in the log for the administrator's reference. In Section 6.3, we will discuss how to choose the default value β for counter c.

5.4 DDoS Attack Stopping Module

Based on the flowchart in Fig. 2, if the flow size is large, the IP variability is high, and the duration is also long, then there is a high possibility that the victim is being attacked by DDoS packets. According to the records in its memory, the controller can analyze the attack's type. Specifically, the IP protocol numbers for TCP, UDP, and ICMP flows are 6, 17, and 1, respectively. If a great deal of records have one certain protocol number (e.g., the percentage of such records is larger than 25%), they are treated as parts of the corresponding attack.

Suppose that the IP address of the victim (linked to port φ_x) is a_v. Depending on the type, the controller installs a flow rule in the corresponding switch to ask it discarding DDoS packets as follows:

- **Flow rule for TCP SYN flood:** eth_type=0x0800, ip_proto=6, tcp_flags=2, ipv4_dst=a_v
- **Flow rule for UDP flood:** eth_type=0x0800, ip_proto=17, ipv4_dst=a_v
- **Flow rule for ICMP flood:** eth_type=0x0800, ip_proto=1, ipv4_dst=a_v

In particular, the term "eth_type=0x0800" means to handle IPv4 packets. When IPv6 is used, we can set the "eth_type" field to 0x86DD and replace the "ipv4_dst" field with the "ipv6_dst" field. Based on the OpenFlow switch specification worked out by ONF [38], the "tcp_flags" field is used to match the flag bits in a TCP header. The length of this field is 2 bytes, and Table 2 gives the meaning of each bit, where bit 0 is the least significant bit. Thus, we add the condition of "tcp_flags=2" for TCP SYN flood, as it means to consider only

Algorithm 4 DDoS Attack Stopping (DAS) Module

```
 1: procedure DAS($\varphi_x$)
 2:     $N_{\mathrm{TCP}} \leftarrow 0$; $N_{\mathrm{UDP}} \leftarrow 0$; $N_{\mathrm{ICMP}} \leftarrow 0$;
 3:     for each record $r_i \in \mathcal{R}_x$ do
 4:         if $w_i = 6$ then
 5:             $N_{\mathrm{TCP}} \leftarrow N_{\mathrm{TCP}} + 1$;
 6:         else if $w_i = 17$ then
 7:             $N_{\mathrm{UDP}} \leftarrow N_{\mathrm{UDP}} + 1$;
 8:         else if $w_i = 1$ then
 9:             $N_{\mathrm{ICMP}} \leftarrow N_{\mathrm{ICMP}} + 1$;
10:         end if
11:     end for
12:     if $N_{\mathrm{TCP}}/|\mathcal{R}_x| \geq 25\%$ then
13:         install the flow rule for TCP SYN flood;
14:     end if
15:     if $N_{\mathrm{UDP}}/|\mathcal{R}_x| \geq 25\%$ then
16:         install the flow rule for UDP flood;
17:     end if
18:     if $N_{\mathrm{ICMP}}/|\mathcal{R}_x| \geq 25\%$ then
19:         install the flow rule for ICMP flood;
20:     end if
21: end procedure
```

SYN packets. Then, the packets satisfying the installed rule will be blocked by the switch to stop attacks.

Algorithm 4 then presents the pseudocode of the DDoS attack stopping module. The code in lines 2 to 11 gathers statistics for each type of packets recorded in $\mathcal{R}_x$, where w_i is the IP protocol number of a record $r_i \in \mathcal{R}_x$. Then, if a certain type of packets occupies more than one quarter of records in $\mathcal{R}_x$, which means that the corresponding DDoS attack is likely happening, the controller then installs the necessary flow rule to stop the attack. This code is given in lines 12–20.

Evidently, the flow rule should be removed once the attack terminates, or new (normal) connections will be also blocked by the switch even if there is no attack. Some studies [28] sets a constant timeout to let the switch remove flow rules after the timeout expires. However, since the duration of attacks is not fixed, the controller may have to repeatedly install the same rule. Moreover, the victim would be still attacked by DDoS packets after timeout (but before the controller installs the rule again). Consequently, instead of using the fixed timeout mechanism, the controller will refer to the amount of residual DDoS traffics to make the decision (i.e., removing flow rules) in our design.

To do so, the switch reports to the controller the amount of traffics blocked by the installed rule. If it falls below a threshold (e.g., 5% of total traffics sent via the switch's port), there is a good possibility that the attack terminates. However, some attacks may be repetitively launched, last for a short time, and then disappear. In this case, the controller will encounter the same problem arisen in the above timeout mechanism. To address this issue, we borrow the notion of *second chance*, which is used in the entry replacement for database systems [39] or main memory [40]. In particular, only when the switch reports that the amount of blocked traffics is below the threshold (i.e., the safe case) for two successive slots, will the controller ask it to remove the flow rule. Like the design in Fig. 5, it can be implemented by using a counter. Initially, the counter is set to two. If the switch reports that the situation is safe, the counter is deducted by one. Otherwise, the counter is reset to two. When the counter becomes zero, the controller infers that the attack terminates and thus asks the switch to remove its flow rules.

5.5 Discussion

One common protocol-based DDoS attack usually produces a mass of packets, originates from many sources (possibly with counterfeit IP addresses), and lasts for a relatively long period. Thus, the flow size, IP variability, and duration will be three important indicators to recognize attacks. In ELD, we first check whether the flow size of a port φ_x overtakes the average value f_{avg}. If so, there may be either attacks or elephant flows. In this case, the IP variability will be a good index to distinguish between them, since elephant flows originate from much fewer sources as compared with DDoS attacks. However, if the IP variability estimating module reports that the variability is pretty high (i.e., exceeding the threshold δ), we refer to both the flow size and duration to distinguish between dangerous DDoS attacks and safe impulse flows. In particular, impulse flows have a flow size smaller than $2f_{\mathrm{avg}}$ and a duration shorter than β slots. In other words, they soon disappear and will not send too many packets to the victim. In this case, even though they are possibly launched by attackers, these impulse flows (or called STLV attacks) will not significantly degrade the victim's performance. As mentioned in Section 5.3, we can ask the controller to simply record this event in a log to reduce its computational cost. Finally, once the controller finds out a suspicious DDoS attack by ELD, it consults the stored records of IP information to find out the attack's type and then installs the corresponding flow rule in the switch to discard following packets sent by the attacker. To prevent the controller from repetitively installing flow rules for the same attack, we adopt a second-chance idea. Only when the switch reports a safe situation for two consecutive slots, will the flow rule be removed. In this way, we can reduce the message overhead for the controller on installing flow rules. The above designs of ELD help the controller fast identify DDoS attacks and stop them with a low cost.

6 Performance Evaluation

The Mininet simulator [41] is used to evaluate the performance of our proposed ELD mechanism. To achieve the functionality of SDN in Mininet, switches are implemented by the Open vSwitch module [42] to support the OpenFlow protocol, and the iPerf tool [43] is adopted to generate practical traffic flows in the network. Moreover, the controller is carried out by the Ryu framework [44] due to two reasons. First, Ryu provides many software components with well-defined application program interfaces (and also extensive documentation), which facilitates program development for the controller. Second, Ryu fully supports all versions of OpenFlow, which includes version 1.5 as of our simulations.

Fig. 1 presents the network topology, where the target network is located in the subnet of 10.0.0.0/24 (in other words, the IP addresses of its hosts are between 10.0.0.0 and 10.0.0.255). There are also three exterior networks located in the subnets of 11.0.0.0/24, 12.0.0.0/24, and 13.0.0.0/24, with which some switches of the target network are connected. Each exterior network contains 150 hosts. In the target network, there are five OpenFlow switches, each linking with 30 hosts. An attacker picks one host in the target network as the victim and manipulates a botnet of 256 members to send DDoS packets to that host. Around 10% of botnet members reside in the target network and others are distributed over the three exterior networks. Both the controller and the victim are not aware of the distribution of botnet members.

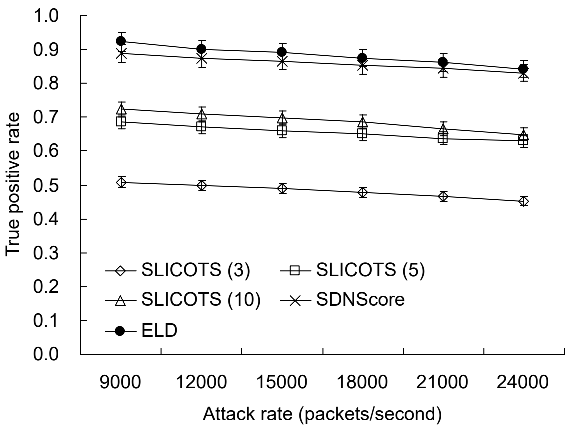
Following the traffic patterns in the work [14], each DDoS flow produces 10000 to 15000 packets per second and lasts for 10 to 20 seconds by default. On the other hand, a normal host transmits 2000 packets every second on the average. A few normal hosts are selected to generate elephant flows and impulse flows, whose destinations are the victim. Each elephant flow behaves similarly with a DDoS flow (in terms of data volume and duration). An impulse flow produces 3000 to 6000 packets in every second and lasts for 2 to 5 seconds. Neither the victim nor the controller has the knowledge of the above settings. The simulation time is set to 1000 seconds.

Below, we measure both the performance and overhead of each SDN-based approach, followed by the comparison of memory utilization between different data storage schemes. After that, we evaluate the effect of parameters in ELD.

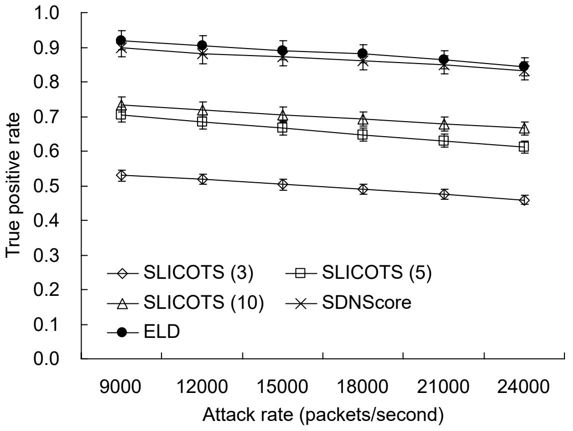
6.1 Comparison on Performance and Overhead

Since the proposed ELD mechanism works on the basis of the SDN technique, we compare it with two DDoS countermeasures also based on SDN, namely SLICOTS [28] and SDNScore [33]. As discussed in Section 3.2, SLICOTS is a lightweight approach which sets a constant timeout T for each flow rule. Thus, a rule will be spontaneously removed by the switch after T seconds. In SLICOTS, we set T to 3 (the default value mentioned in the study [28]), 5, and 10 seconds to observe its effect. On the other hand, SDNScore is a much more sophisticated approach by checking eight features of packets. There are 28 tables built in SDNScore to help the controller grade every packet, so as to decide whether to send or drop the packet. Evidently, SLICOTS aims to reduce the message overhead of the controller while SDNScore seeks to improve the accuracy on attack recognition. By comparing both of them, we can demonstrate how our ELD mechanism strikes a good balance between performance and overhead on DDoS defense. In ELD, we set δ to 80% and β to 2, and the length of each slot is 3 seconds.

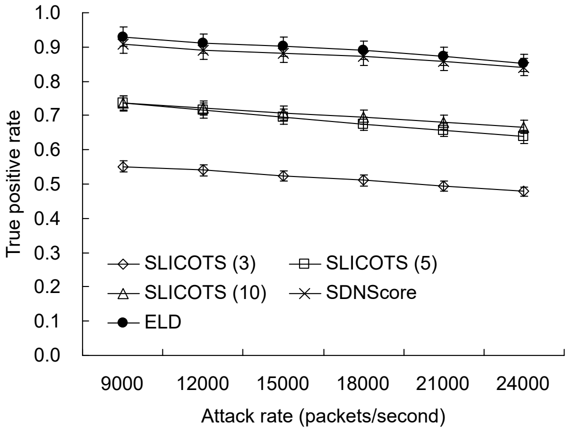
Fig. 7 gives the *true positive rate* of each approach, which is defined by the ratio of the successfully blocked DDoS packets to the total DDoS packets produced by an attack. In this experiment, there are 30 DDoS attacks launched, including 10 TCP SYN flood, 10 UDP flood, and 10 ICMP flood attacks. The interval between two adjacent attacks is set to [15, 20] seconds. The attack rate is increased from 9000 to 24000 packets per second. Generally speaking, the true positive rate of each approach decreases when the attack rate increases, as the attacker sends more packets to the victim within the same attacking period (i.e., 10–20 seconds). The true positive rate of SLICOTS is significantly improved by extending the timeout from 3 to 5 seconds. The reason is that the switch will not block DDoS packets after removing the flow rule and before the controller reinstalling it. Thus, a longer timeout helps SLICOTS increase its true positive rate. However, even though we increase the timeout to 10 seconds, SLICOTS can block at most 73.6% of DDoS packets. On the other hand, SDNScore considers more features of packets and also their relationship, so it can substantially increase the true positive rate as compared with SLICOTS. Our ELD mechanism recognizes DDoS attacks based on flow size, IP variability, and duration. In contrast with SDNScore, it refers to much fewer features. However, ELD can still perform as well as SDNScore, which verifies its effectiveness on attack detection and reaction.



(a) TCP SYN flood



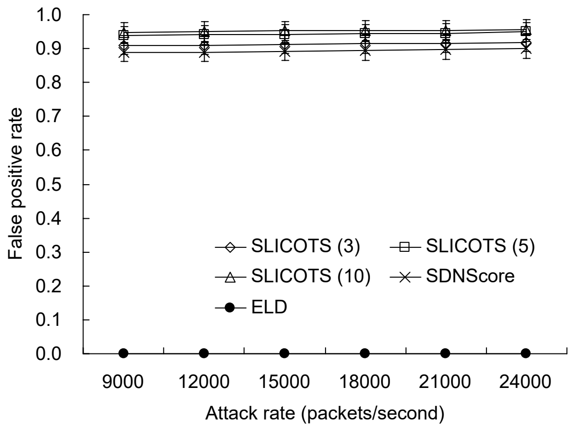
(b) UDP flood



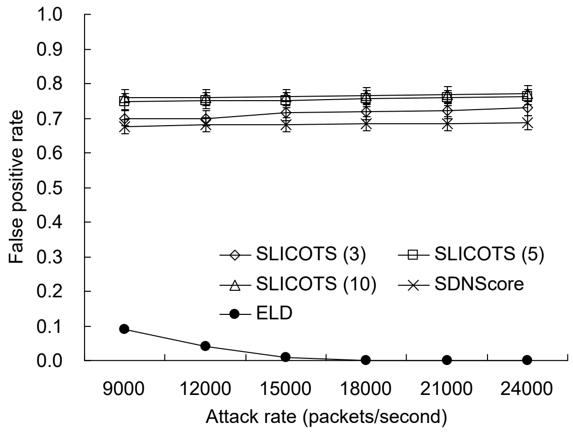
(c) ICMP flood

Fig. 7: Comparison on the true positive rate.

Fig. 8 compares the *false positive rate* of SLICOTS, SDNScore, and ELD, which is defined by the ratio of the blocked packets of elephant flows and impulse flows to the total packets that they produce. In the experiment, there are 10 elephant flows and 10 impulse flows generated. Since both SLICOTS and SDNScore do not discriminate between elephant/impulse flows and DDoS ones, they will result in many false alarms. In particular, SLICOTS blocks more than 91% and 70% of packets of elephant flows and impulse flows, respectively. SDNScore



(a) Elephant flows



(b) Impulse flows

Fig. 8: Comparison on the false positive rate.

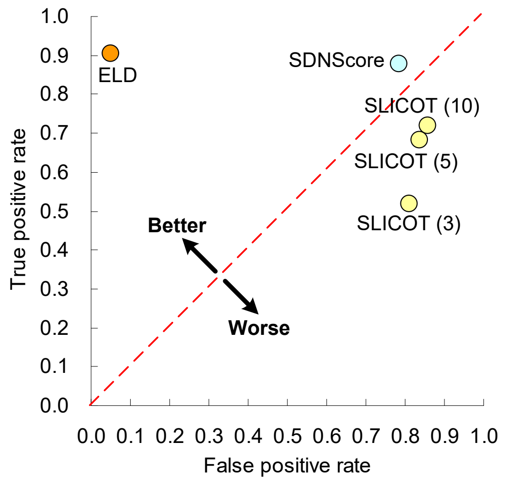


Fig. 9: The ROC chart.

TABLE 3: Comparison on the number of installed flow rules (with 95% confidence).

Approaches	Installed rules
SLICOTS (3)	161 ± 0.116
SLICOTS (5)	101 ± 0.069
SLICOTS (10)	57.3 ± 0.031
SDNScore	60 ± 0
ELD	30 ± 0

blocks more than 89% and 68% of packets of elephant flows and impulse flows, respectively. Since elephant flows originate from just few sources, it is easy for ELD to distinguish them from DDoS attacks by referring to the IP variability, so ELD blocks almost no packets of elephant flows (i.e., zero false alarm). On the other hand, when the attack rate is small, a few impulse flows may behave similar to DDoS flows (i.e., producing too many packets). In this case, they would be treated as DDoS ones and thereby blocked by ELD. However, such false alarms can be eliminated in ELD by increasing the attack rate, as shown in Fig. 8(b).

By considering 30 DDoS attacks, 10 elephant flows, and 10 impulse flows as in the previous two experiments, we then use a *receiver operating characteristic (ROC)* chart in terms of both false positive rate and true positive rate to compare the accuracy of each approach. In this experiment, each DDoS flow produces 10000 to 15000 packets per second. Fig. 9 presents the ROC chart, where the diagonal line indicates the *random guess* line. When the result of an approach is closer to the upper left corner, it means that the approach has higher accuracy. For SLICOTS, increasing the timeout T helps improve its accuracy. However, the effect of improvement is insignificant by increasing T from 5 to 10 seconds, which shows the limit of SLICOTS. On the other hand, the result of SDNScore lies close to the random guess line, which implies that the accuracy of SDNScore is slightly higher than 50%. The result of our ELD mechanism shows the best predictive power among all approaches, where it not only keeps a high positive rate (for correctly recognizing DDoS attacks) but also has a lower false positive rate (for distinguishing elephant/impulse flows from attacks).

Next, we measure the number of flow rules installed by the controller to defend against DDoS attacks. In the experiment, there are 30 attacks launched in sequence, each with a duration of [10, 20] seconds. When an attack terminates, the next attack will be launched after 15 to 20 seconds. Table 3 gives the average number of flow rules installed, where we repeat the simulation for 10000 times and the confidence level is set to 95%. Obviously, the more flow rules are installed, the higher message overhead will be incurred by the controller. Recall that SLICOTS sets a constant timeout (i.e., T seconds) for each installed rule. Let $D_i^{\mathbf{A}}$ be the duration of the i-th attack. Then, the total number of flow rules installed in SLICOTS is calculated by $\sum_{i=1}^{30} \lceil D_i^{\mathbf{A}}/T \rceil$. Expectably, a shorter timeout leads the controller to install flow rules more frequently, as each rule will be soon removed by the switch. On the other hand, SDNScore decides whether to drop or forward each packet by consulting its tables. Thus, the controller needs to install two flow rules for every attack. That is why the number of flow rules installed in SDNScore is the double of the number of attacks. As discussed in Section 5.4, our ELD mechanism installs just one flow rule to drop the packets generated by each attack. The controller will remove the rule only when it finds that the attack terminates (by the second-chance method). Thus, the number of flow rules installed in ELD will be equal to the number of attacks, which results in the lowest message overhead. In particular, ELD can save 81.4%, 70.3%, 47.6%, and 50.0% of flow rules to be installed, as compared with SLICOTS (3), SLICOTS (5), SLICOTS (10), and SDNScore, respectively.

After that, we evaluate the computational cost of the controller (in terms of the number of operations) to detect and stop DDoS attacks. Let N_{total} be the total number of packets. For SLICOTS, it records four features of each packet (i.e., source IP address, destination IP address, source port, destination port), which requires $4N_{\text{total}}$ operations to conduct feature

TABLE 4: Comparison on the computational cost of the controller.

Approaches	Feature extraction	Attack identification	Extra actions	Total costs
SLICOTS	4 features/packet	4 operations/check	None	400,080
SDNScore	8 features/packet	28 operations/packet	1 operation/packet	3,700,000
ELD	1 feature/packet	3 operations/check	1 operation/check	100,080

extraction of packets. Suppose that SLICOTS checks whether an attack occurs every N_{check} packets on the average. Then, it takes at most $4\lceil N_{\text{total}}/N_{\text{check}}\rceil$ operations to perform attack identification [28]. Thus, the total number of operations used in SLICOTS will be

$$N_{\text{SLICOTS}} = 4N_{\text{total}} + 4\lceil N_{\text{total}}/N_{\text{check}}\rceil. \quad (7)$$

As discussed in Section 3.2, SDNScore extracts eight features from each packet. To find out an attack, SDNScore will refer to 28 tables to grade every packet. Moreover, it has to decide whether to drop or send the packet. Therefore, the total number of operations required in SDNScore is estimated by

$$N_{\text{SDNScore}} = 8N_{\text{total}} + 28N_{\text{total}} + N_{\text{total}} = 37N_{\text{total}}. \quad (8)$$

In ELD, it records only the source IP address of each packet. Like SLICOTS, ELD also checks if an attack occurs every N_{check} packets. Based on the flowchart in Fig. 2, there are three operations involved to perform attack identification (i.e., checking flow size, IP variability estimating module, and impulse flow checking module). Besides, the controller will check whether to remove flow rules every N_{check} packets after it finds out an attack (and installs flow rules to stop the attack). Consequently, the total number of operations taken in ELD is measured by

$$\begin{aligned} N_{\text{ELD}} &= N_{\text{total}} + 3\lceil N_{\text{total}}/N_{\text{check}}\rceil + \lceil N_{\text{total}}/N_{\text{check}}\rceil \\ &= N_{\text{total}} + 4\lceil N_{\text{total}}/N_{\text{check}}\rceil. \end{aligned} \quad (9)$$

Table 4 compares the computational cost spent by the controller in these three approaches, where we set $N_{\text{total}} = 100000$ and $N_{\text{check}} = 5000$. In particular, SLICOTS and SDNScore have around 4 and 37 times of the number of required operations than ELD, which shows that ELD can greatly reduce the controller's overhead in computation.

6.2 Comparison on Memory Utilization

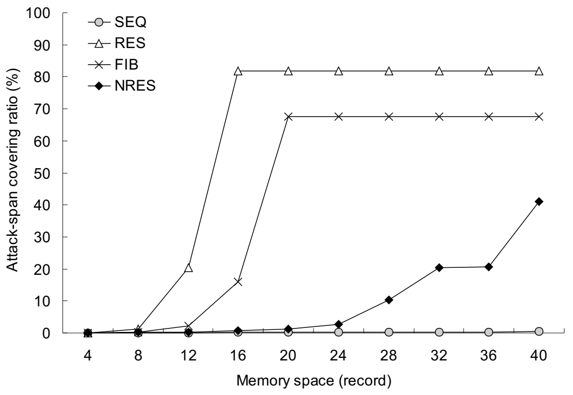
As discussed in Section 5.1, the previous RES scheme [34] will store many outdated records when the memory space becomes large. Thus, the NRES scheme is developed to conquer the problem. In this section, we study their memory utilization. Except for RES, two additional data storage schemes are also considered for comparison:

- *Sequential (SEQ) storage scheme:* SEQ stores records just like a FIFO (first-in, first-out) queue.
- *Fibonacci (FIB) storage scheme:* FIB stores records based on a Fibonacci sequence, where $F_0 = 0$, $F_1 = 1$, and $F_k = F_{k-1} + F_{k-2}$ for any $k > 1$. In an ideal case, FIB will keep records with ages 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, and so on.

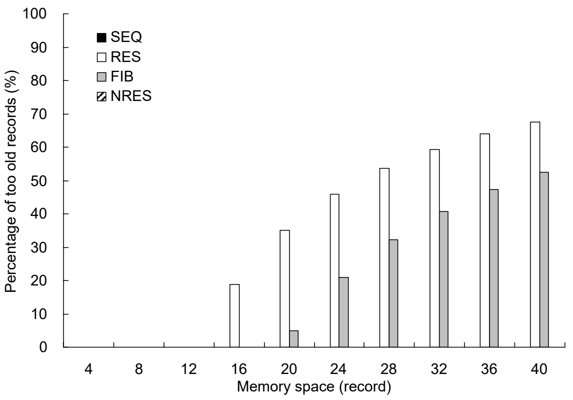
Table 5 lists the oldest record (in age) that can be kept by each data storage scheme, where the maximum number of records stored in the memory is increased from 4 to 40. For SEQ, the age of the oldest record is linear with the memory size, which means that it cannot store records long time ago. At the other extreme, the oldest records in RES and FIB have very large ages. This phenomenon implies that both RES and FIB will keep too old (i.e., useless) records when the memory space is relatively large. Compared with these schemes, the age-growing speed of the oldest record in NRES is reasonable, so it can keep enough historical records while removing useless records.

TABLE 5: The oldest records kept by different data storage schemes.

Space	SEQ	RES	FIB	NRES
4	4	8	5	5
8	8	128	34	17
12	12	2,048	233	34
16	16	32,768	1,597	67
20	20	524,288	10,946	131
24	24	8,388,608	75,025	259
28	28	134,217,728	514,229	1,026
32	32	2,147,483,648	3,524,578	2,049
36	36	34,359,738,368	24,157,817	2,053
40	40	549,755,813,888	165,580,141	4,099



(a) Attack-span covering ratio



(b) Percentage of too old records

Fig. 10: Comparison on memory utilization.

Fig. 10 compares the memory utilization of SEQ, RES, FIB, and NRES in terms of recording DDoS packets, where the attack produces 10000 packets. In particular, Fig. 10(a) gives the *attack-span covering ratio*, which is defined as follows:

$$\frac{\text{largest age of records within attack's lifespan}}{\text{the number of packets produced by the attack}} \times 100\%. \quad (10)$$

For example, if at most 16 records can be stored in the memory, RES ideally keeps records with ages 1, 2, 4, $\cdots$, 8192, 16384, and 32768. Thus, the largest age of records within the attack's lifespan is 8192, and the attack-span covering ratio is 81.92%. Since each scheme must store the record with age 1, a higher ratio means that the scheme has a broader view of the attack. Based on the experimental data in Fig. 10(a), the ratio of SEQ is pretty close to zero, which means that SEQ is only able to store very recent records and thus has a quite narrow view of the attack. On the contrary, both SEQ and RES have evidently higher ratios, which implies that they can keep records in the early stage of the attack, thereby expanding their views of the attack. The attack-span covering ratio of NRES significantly grows as the memory space is enlarged (especially when the memory can store more than 24 records), so it also has a much broader view than SEQ.

On the other hand, Fig. 10(b) shows the percentage of *too old records*, which is defined by the percentage of the records whose timestamps are earlier than the beginning of the attack. When the memory can store more than 16 records, both RES and FIB start keeping too old records, and their percentages grow very fast as the memory's size increases. This phenomenon implies that RES and FIB actually store much fewer records for DDoS packets than other schemes. In this case, they will substantially decrease the resolution of view for the attack.

From the result in Fig. 10, our NRES scheme has a much higher attack-span covering ratio than SEQ, and will not store too old records as compared with both RES and FIB. Therefore, NRES can provide a broad and high-resolution view of potential attacks, which facilitates the recognition of DDoS packets for the controller.

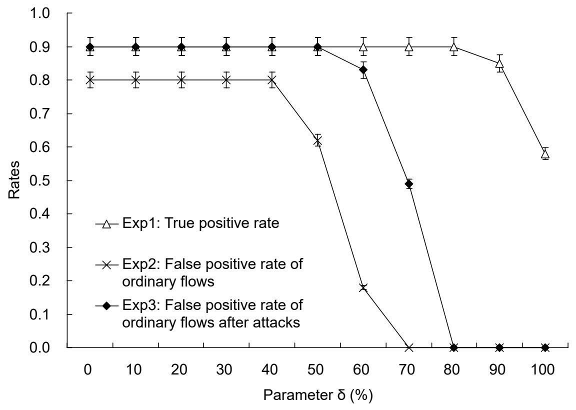


Fig. 11: Effect of parameter δ on the true positive rate and the false positive rate.

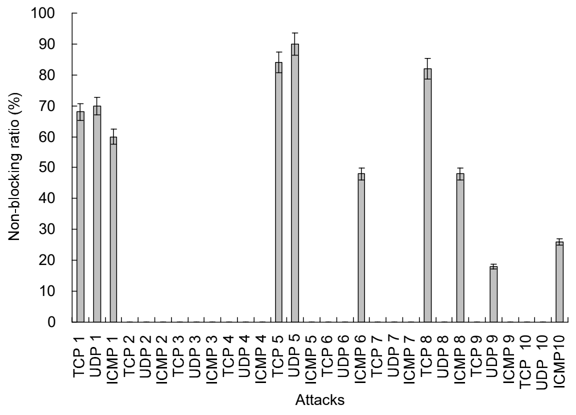


Fig. 12: Non-blocking ratios of attacks by setting $\beta = 1$.

6.3 Effect of Parameters in ELD

Finally, we investigate the effect of parameters δ and β on ELD's performance. Specifically, when the IP variability V_x in Eq. (6) overtakes the threshold δ, the controller infers that there is a high possibility of attacks to the victim. To find an appropriate value for δ, three experiments are conducted as follows:

- **Experiment 1:** Totally 30 non-overlapped DDoS attacks are launched during 1000 seconds, where the interval between two adjacent attacks is set to [15, 20] seconds. Each attack produces [10000, 15000] packets every second and has lifespan of [10, 20] seconds. In this experiment, we evaluate the true positive rate.
- **Experiment 2:** There is no DDoS attack in the 1000-second simulation. Each host generates one ordinary flow which produces [1500, 2000] packets per second. We measure the false positive rate of these flows, which is defined by the ratio of the blocked packets to their produced packets.
- **Experiment 3:** It is a combination of both experiments 1 and 2 (i.e., DDoS and ordinary flows coexist). In the experiment, we observe the false positive rate of ordinary flows after each attack.

Fig. 11 presents the experimental data, where the value of δ is varied from 0% to 100%. In experiment 1, since a DDoS attack usually originates from multiple sources, a lower δ value can help the controller find out more packets of the attack. Therefore, the true positive rate is kept in 0.9 when δ is increased from 0% to 80%. However, because DDoS packets could be also produced from the same sources, some of them may be thus mistaken for normal packets when δ is set to too large. That is why the true positive rate drops when δ is larger than 80%.

In experiment 2, a smaller δ value contrarily leads to a higher false positive rate. In particular, when δ is below 40%, the false positive rate is kept in 0.8. The reason is that there are a number of hosts generating these ordinary flows, which raises their IP variability. As δ increases, some low-volume flows will be ignored by the controller (so their packets will not be blocked), thereby reducing the false positive rate. When δ overtakes 70%, the controller can recognize all ordinary flows, so there will be no false alarm accordingly.

Similarly, a smaller δ value also keeps a higher false positive rate in experiment 3. Since DDoS and ordinary flows coexist, the controller has to raise the value of δ to distinguish between them. As can be seen from Fig. 11, the controller can eliminate false alarms when $\delta \geq 80\%$. Based on the results of these three experiments, we suggest setting threshold δ in [80%, 90%] to increase the true positive rate and decrease the false positive rate.

On the other hand, parameter β is used in the impulse flow checking module (referring to Section 5.3). In particular, when the flow size keeps above the average value f_{avg} for β consecutive slots, the controller infers that a DDoS attack may be happening. Otherwise, it could be an impulse flow. To find a suitable value of β, we also launch 30 DDoS attacks as in experiment 1 and measure the *non-blocking ratio*, which is defined by the ratio of the packets not blocked by the controller to the total packets produced by an attack. Fig. 12 gives the experimental data by setting $\beta = 1$, where "TCP i", "UDP i", and "ICMP i", $i = 1..10$, denote the i-th TCP SYN flood, UDP flood, and ICMP flood attacks, respectively. It can be observed that some attacks still have pretty high non-blocking ratios. However, when β is set to two (or larger), the controller can recognize each attack and block its packets accordingly. Here, we omit the result of $\beta = 2$ because all attacks have zero non-blocking ratios. Since parameter β should be set as small as possible to prevent the controller from checking too many slots, a better value of β will be thus 2 in this experiment.

7 Conclusion

DDoS attacks compel victim servers to be unable to provide services to legitimate users by overwhelming victims with lots of requests. They would also greatly degrade SDN's performance, because the controller has to deal with many connections generated by the attacks. Thus, this paper develops a lightweight but efficient ELD mechanism to defend against protocol-based DDoS attacks, whose objective is to reduce the controller's overhead to fast recognize attacks and stop them in time. Since the memory space is limited, we propose an NRES data storage scheme in ELD to help the controller record more recent packets while keeping a few ancient packets for reference, so as to offer a broad and high-resolution view of attacks. By checking the signatures of flow size, IP variability, and duration, ELD differentiates normal flows from DDoS ones, thereby avoiding dropping the packets of elephant flows and impulse flows. Simulation results show that ELD increases the true positive rate, substantially decreases false alarms, and significantly reduces the controller's overhead, as compared with both SLICOTS and SDNScore. Moreover, our proposed NRES scheme outperforms other data storage schemes in terms of memory utilization, which helps the controller store packets' information and detect DDoS attacks more efficiently.

References

[1] N. Hoque, D.K. Bhattacharyya, and J.K. Kalita, "Botnet in DDoS attacks: trends and challenges," *IEEE Comm. Surveys & Tutorials*, vol. 17, no. 4, pp. 2242–2270, 2015.

[2] Z. Trabelsi, S. Zeidan, and K. Hayawi, "Denial of firewalling attacks (DoF): the case study of the emerging BlackNurse attack," *IEEE Access*, vol. 7, pp. 61596–61609, 2019.

[3] D. Kreutz, F.M.V. Ramos, P.E. Verissimo, C.E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: a comprehensive survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[4] Y.C. Wang and S.Y. You, "An efficient route management framework for load balance and overhead reduction in SDN-based data center networks," *IEEE Trans. Network and Service Management*, vol. 15, no. 4, pp. 1422–1434, 2018.

[5] T. Zhu, D. Feng, F. Wang, Y. Hua, Q. Shi, J. Liu, Y. Cheng, and Y. Wan, "Efficient anonymous communication in SDN-based data center networks," *IEEE/ACM Trans. Networking*, vol. 25, no. 6, pp. 3767–3780, 2017.

[6] J.H. Cox, R. Clark, and H. Owen, "Leveraging SDN and WebRTC for rogue access point security," *IEEE Trans. Network and Service Management*, vol. 14, no. 3, pp. 756–770, 2017.

[7] Y.C. Wang and H. Hu, "An adaptive broadcast and multicast traffic cutting framework to improve Ethernet efficiency by SDN," *J. Information Science and Engineering*, vol. 35, no. 2, pp. 375–392, 2019.

[8] P. Dong, X. Du, H. Zhang, and T. Xu, "A detection method for a novel DDoS attack against SDN controllers by vast new low-traffic flows," *Prof. IEEE Int'l Conf. Comm.*, 2016, pp. 1–6.

[9] W. Wang, Y. Sun, K. Salamatian, and Z. Li, "Adaptive path isolation for elephant and mice flows by exploiting path diversity in datacenters," *IEEE Trans. Network and Service Management*, vol. 13, no. 1, pp. 5–18, 2016.

[10] M. Bogdanoski, T. Shuminoskiand, and A. Risteski, "Analysis of the SYN flood DoS attack," *Int'l J. Computer Network and Information Security*, vol. 8, pp. 1–11, 2013.

[11] K. Treseangrat, S.S. Kolahi, and B. Sarrafpour, "Analysis of UDP DDoS cyber flood attack and defense mechanisms on Windows Server 2012 and Linux Ubuntu 13," *Proc. IEEE Int'l Conf. Computer, Information and Telecomm. Systems*, 2015, pp. 1–5.

[12] P. Harshita and R. Nayyar, "Detection of ICMP flood DDoS attack," *Int'l J. Computer Science Trends and Technology*, vol. 5, no. 2, pp. 199–205, 2017.

[13] C.Y. Lin, C. Chen, J.W. Chang, and Y.H. Chu, "Elephant flow detection in datacenters using OpenFlow-based hierarchical statistics pulling," *Proc. IEEE Global Comm. Con.*, 2014, pp. 2264–2269.

[14] S.K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey, "Bohatei: flexible and elastic DDoS defense," *Prof. USENIX Security Symposium*, 2015, pp. 817–832.

[15] Y.C. Wang and H. Hu, "A Low-cost, high-efficiency SDN framework to diminish redundant ARP and IGMP traffics in large-scale LANs," *Prof. IEEE Computer Software and Applications Conf.*, 2018, pp. 894–903.

[16] P. Skrobanek, *Intrusion Detection Systems*. London: IntechOpen, 2011.

[17] D. Pengfule, T. Zhihong, Z. Hongli, W. Yong, Z. Liang, and G. Sanchuan, "Detection and defense of SYN flood attacks based on dual stack network firewall," *Prof. IEEE Int'l Conf. Data Science in Cyberspace*, 2016, pp. 526–531.

[18] G. Yao, J. Bi, and A.V. Vasilakos, "Passive IP traceback: disclosing the locations of IP spoofers from path backscatter," *IEEE Trans. Information Forensics and Security*, vol. 10, no. 3, pp. 471–484, 2015.

[19] S. Savage, D. Wetherall, A. Karlin, and T. Anderson, "Practical network support for IP traceback," *ACM SIGCOMM Computer Comm. Review*, vol. 30, no. 4, pp. 295–306, 2000.

[20] C. Gong and K. Sarac, "A more practical approach for single-packet IP traceback using packet logging and marking," *IEEE Trans. Parallel and Distributed Systems*, vol. 19, no. 10, pp. 1310–1324, 2008.

[21] A. Izaddoost, M. Othman, and M.F.A. Rasid, "Accurate ICMP traceback model under DoS/DDoS attack," *Proc. Int'l Conf. Advanced Computing and Comm.*, 2007, pp. 441–446.

[22] B. Al-Duwairi and G. Manimaran, "Intentional dropping: a novel scheme for SYN flooding mitigation," *Proc. IEEE INFOCOM*, 2005, pp. 2820–2824.

[23] B. Hang and R. Hu, "A novel SYN cookie method for TCP layer DDoS attack," *Proc. Int'l Conf. Future BioMedical Information Engineering*, 2009, pp. 445–448.

[24] Understanding unicast reverse path forwarding. [Online]. Available: https://www.cisco.com/c/en/us/about/security-center/unicast-reverse-path-forwarding.html

[25] R. Xu, W. L. Ma, and W. L. Zheng, "Defending against UDP flooding by negative selection algorithm based on eigenvalue sets," *Proc. Int'l Conf. Information Assurance and Security*, 2009, pp. 342–345.

[26] P. Rengaraju, V.R. Ramanan, and C.H. Lung, "Detection and prevention of DoS attacks in software-defined cloud networks," *Proc. IEEE Conf. Dependable and Secure Computing*, 2017, pp. 217–223.

[27] P. Iannucci and M. Gupta, *IBM SmartCloud: building a cloud enabled data center*. Endicott: IBM Redbooks, 2013.

[28] R. Mohammadi, R. Javidan, and M. Conti, "SLICOTS: an SDN-based lightweight countermeasure for TCP SYN flooding attacks," *IEEE Trans. Network and Service Management*, vol. 14, no. 2, pp. 487–97, 2017.

[29] UDP-based amplification attacks. [Online]. Available: https://www.us-cert.gov/ncas/alerts/TA14-017A

[30] L. Mutu, R. Saleh, and A. Matrawy, "Improved SDN responsiveness to UDP flood attacks," *Proc. IEEE Conf. Comm. and Network Security*, 2015, pp. 715–716.

[31] H.C. Wei, Y.H. Tung, and C.M. Yu, "Counteracting UDP flooding attacks in SDN," *Proc. IEEE Conf. Network Softwarization*, 2016, pp. 367–371.

[32] Y. Yu, L. Guo, Y. Liu, J. Zheng, and Y. Zong, "An efficient SDN-based DDoS attack detection and rapid response platform in vehicular networks," *IEEE Access*, vol. 6, pp. 44570–44579, 2018.

[33] K. Kalkan, G. Gur, and F. Alagoz, "SDNScore: a statistical defense mechanism against DDoS attacks in SDN environment," *Proc. IEEE Symp. Computers and Comm.*, 2017, pp. 669–675.

[34] Y.C. Wang, Y.Y. Hsieh, and Y.C. Tseng, "Multiresolution spatial and temporal coding in a wireless sensor network for long-term monitoring applications," *IEEE Trans. Computers*, vol. 58, no. 6, pp. 827–838, 2009.

[35] Transmission Control Protocol. [Online]. Available: https://tools.ietf.org/html/rfc793

[36] The Addition of Explicit Congestion Notification (ECN) to IP. [Online]. Available: https://tools.ietf.org/html/rfc3168

[37] Robust Explicit Congestion Notification (ECN) Signaling with Nonces. [Online]. Available: https://tools.ietf.org/html/rfc3540

[38] OpenFlow. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

[39] C.C. Lo and K.L. Sue, “Second chance replacement policy for mobile database overflow,” *Proc. IEEE Global Telecomm. Conf.*, 2002, pp. 1683–1687.
[40] A. Silberschatz, P.B. Galvin, and G. Gagne, *Operating System Concepts*. Hoboken: Wiley, 2008.
[41] Mininet. [Online]. Available: http://mininet.org/
[42] Open vSwitch. [Online]. Available: https://www.openvswitch.org/
[43] iPerf. [Online]. Available: https://iperf.fr/
[44] Ryu. [Online]. Available: https://osrg.github.io/ryu/