TECHNISCHE UNIVERSITÄT CHEMNITZ

Faculty of Communication Networks

Department of Electrical Engineering and Information Technology

Research Project

Source-based Intelligent Detection of Low and Slow DDoS Attacks in Software Defined Networks

T M Rayhan Gias

Chemnitz, 30th January 2019

Supervisor:	M. Sc. Trung Phan Van
Advisor:	Prof. Dr.-Ing. Thomas Bauschert

Abstract

Manageability, Scalability, Controllability, and Dynamism of the entire network intrigue data centers and cloud service providers to adopt Software Defined Networks(SDN) as their principal technology. Meanwhile, the current SDN architecture brings new vulnerabilities as attackers try to exploit the functionalities of network components and disrupt the services of the cloud in a stealth mode. Distributed Denial of Service (DDoS) attack is one of the main network security problems over the last decade. The newest addition to this problem is Low and Slow Distributed Denial of Service (LSDDoS). It sends minimum rate of a packet to disrupt the services of server and overflow flow tables of SDN switches which can collapse the SDN controller and the whole network as a consequence. On the other hand, the capabilities of SDN, including software-based traffic analysis, centralized control, global view of the network make it easier to detect and react to LSDDoS attacks.

In this research project, we have introduced an intelligent and novel approach to point out the attack source precisely. Self Organizing Map (SOM), an unsupervised machine learning approach, is used to classify the attack due to its high accuracy and capability of reliable prediction based on their neurons. We trained our algorithm in such a way that it can detect LSDDoS attack with its source within a short span of time. In the introduced scheme, the calculation of proposed parameters is done by the statistic information of flow rules which lead to the server. Then it is fed to the classification algorithm to make a final decision. We have trained and tested our algorithm by using the real-time experiment data from the emulated environment. The main challenge of attack detection is to detect it within a short span of time with high accuracy. Our performance evaluation shows that our detection module can perform as per requirement. Detection time and false alarm rate of our proposed approach are very low which represents its effectiveness. Thorough practical experiments are conducted in the Software Defined Networking environment, it is proved that the proposed classification and the novel mechanism can be an effective and innovative approach to face LSDDoS attacks, protect the SDN switches and the SDN controller from being damaged.

Contents

SDN	**S**oftware **D**efined **N**etworks
DoS	**D**enial of **S**ervice
DDoS	**D**istributed **D**enial of **S**ervice
LSDDoS	**L**ow and **S**low **D**istributed **D**enial of **S**ervice
ML	**M**achine **L**earning
SOM	**S**elf **O**rganizing **M**ap
TCP	**T**ransmission **C**ontrol **P**rotocol
HTTP	**H**yper **T**ext **T**ransfer **P**rotocol
TCAM	**T**ernaray **C**ontent **A**ddressable **M**emory
OF	**O**penFlow
ONF	**O**pen **N**etworking **F**oundation
ONOS	**O**pen **N**etworking **O**perating **S**ystem
API	**A**pplication **P**rogrammable **I**nterface
REST	**R**epresentational **S**tate **T**ransfer
LLDP	**L**ink **L**ayer **D**iscovery **P**rotocol
ARP	**A**ddress **R**esolution **P**rotocol
IPV4	**I**nternet **P**rotocol **V**ersion 4
RTT	**R**ound **T**rip **T**ime
Wget	**W**ide web and **get**
FAR	**F**alse **A**larm **R**ate

1 Introduction

In this information age, connecting billions of computers worldwide internet is the most essential communication medium. In another note, smart devices are getting widespread at lower prices and a large number of legitimate clients are connecting to the internet. As they are now an essential part of our everyday life, application servers on the internet must be kept secured and always alive to serve. But this permeant connectivity among computers and other smart devices attracts unwanted malicious users to disrupt the services of the application server by launching **"Denial of Service"(DoS)**.

Network security, in general, is defined by four specific properties which are: validity, consistency, authenticity, and availability [1]. These properties must be guaranteed at all costs by taking appropriate defensive measures from the physical layer up to the application layer. Network security is also facing the challenge to ensure the host security because of the pervasive nature of the internet. Indeed, physical, networking and operating system level security measures can only be applied to relatively small administrative domains which are limiting the practical measures to the transport and application layers.

Distributed Denial of Service (DDoS) attack is one of the dangerous threats for services on the internet. In 2018, DDoS attacks were set at 1.7 Tbps [2] which is the largest attack so far. On the other hand, detection and taking appropriate measures are lacking far behind. Successfully detecting the malicious users among the legitimate users is very difficult. Mitigation systems which are available require the victim's assistance or the victim administrators to be active when an attack occurs.

DDoS attacks are now evolving in some serious manner. At the earlier stage, it requires huge resources to initiate an attack. Lots of computers (or botnets) are needed to be connected to disrupt the services of different application servers. In March 2013, most bandwidth intensive DDoS attack has been launched against an anti-spam company, Spamhaus, and it also affected the global internet performance [3]. At present, DDoS attacks are getting much more harmful as **'Low and Slow DDoS (LSDDoS)'** are introduced to the world.

LSDDoS attacks work differently from the traditional flooding-based DoS attack. The common goal of all LSDDoS is denying an internet service by sending a relatively low quantity of bytes during an extended period of time with a specific interval. While a flooding-based attack cannot be executed since the resources it requires are not accessible by everyone. But LSDDoS is more affordable and can even be launched from a single machine with a limited internet connection. LSDDoS attacks always try to acquire all the connections provided by the server.

Separating the intelligence from the data plane to control plane, **Software-Defined Networks (SDN) [4]** flourishes as a promising network framework that allows programmability, flexible control, and agile management. Several technology giants like Google, Amazon, Microsoft have already deployed SDN in their networks. SDN controller which acts as a brain of SDN network monitors and controls network traffic which flows through OpenFlow switches. In addition, OpenFlow **[4], [5]** is the first standard communication protocol between the controller and the switches which researchers have developed in recent years. It provides the SDN controller and OpenFlow switches with messages **[6]** for communication such as: packet-received, send-packet-out, modify-forwarding-table, get-stats etc. An OpenFlow switch has its own data path which describes a clear flow-table abstraction and the flow-entries include several packet fields to match and instructions such as send-out-port or drop. Large number of SDN applications have been developed to enable various network functionalities such as dynamic flow scheduling **[7]**, holistic network monitoring, management **[8]** and security mechanism deployment in large networks **[9]**.

SDN design itself, regrettably, has serious security problems. Specifically, The SDN data plane is not allowed to take any decisions for the outgoing data flow. SDN switches are vulnerable to flow table overflow. In addition, SDN-enabled switches only support a small number of flow rules, e.g. 2000-3000 flow rules. One of the notable vulnerabilities is flow rules are stored in power-hungry and expensive Ternary Content Addressable Memory (TCAM). This limited space of TCAM memory can easily be flooded with fake flow rules. Data flow that does not match with installed flow rules in the switch will generate "packet-in" messages. These messages will initiate a query to the SDN controller for the new flow rules and attackers use this mechanism to install new rules forcefully. This leads to the fact that not only a server or a target network may become the LSDDoS victim, but also the SDN controller and switches may stop working because of resource exhaustion. Accordingly, legitimate users or flows are not able to reach the network while network resources are suffered from exhaustion. Therefore, damaging effects cause great difficulties to tackle LSDDoS attacks in Software Defined Networks.

Increasing damage to service providers on the internet makes the detection and distinguish between a legitimate and a malicious client high priority. In a network, where network provider, server provider and host provider are often different entities without direct access to each other's infrastructure, defense against these attacks can be tricky. The host provider cannot reliably defend against the attacks that make it impossible for the hosts to be reached in the first place. It is necessary to defend against the attacks within the network as close to the attacker as possible. The network provider, on the other hand, might not even be aware of an attack targeting the victim server that solely drains the server resources. Software Defined Networks come as a blessing to offer various unique features that make it easier to build a defense system against LSDDoS attacks. The separation of control plane and data plane simplifies implementation, testing, and deployment of defense mechanisms in the network.

For large scale LSDDoS attacks, existing defense mechanism can only detect the presence of the attack **[10]** - **[17]**. However, the performance is ineffective in differentiating between legitimate and malicious users. In this work, we are proposing a flow-based detection engine which is developed using Machine Learning (ML) based approach **"Self-Organizing Map" (SOM)** to detect the LSDDoS attacker. This approach exploits the fact that LSDDoS flows actively induce network congestion whereas normal flows actively avoid network congestion. Usually, TCP or HTTP flows send the similar number of packets and after a certain time interval, the flows will be stopped. But LSDDoS sends very few packets and will try to keep the connection alive. Keeping these basic feature of LSDDoS attack in mind, we have analyzed the traffic flow of both normal user and attacker in SDN environment and extracted necessary statistics for our proposed ML approach.

The rest of the paper is structured as follows. **Chapter 2** reviews the state-of-the-art and related works. **Chapter 3** contains the background and LSDDoS attack scenario, followed by an introduction of Machine Learning approach for our detection engine in **Chapter 4**. **Chapter 5** details the LSDDoS attack simulation and **Chapter 6** describes our results and the performance evaluation. **Chapter 7** concludes the paper and discusses future work.

2 Related Work

Precise detection of LSDDoS is the main obstacle in the network security realm which intrigues lots of researchers in SDN research community. Based on several parameters, (such as: idle timeout of the OpenFlow protocol, the number of unpaired flow rules, rule installation rate in the SDN switches, TCAM memory consumption rate) the defense mechanism will trigger. Author of **[10]** proposed a policy to drop flow rules from the SDN switches randomly during attack though there is a probability of dropping the flow rules of SDN siwtches which belongs to legitimate clients. Another approach is to generate artificial jitter and set a dynamic timeout when an attacker wants to probe idle and hard timeout of SDN switches for crafting LSDDoS packets **[11]**. Now controllers have to process more packet-in messages and extra forwarding delays for the benign packets. Monitoring and scoring system for the suspicious behavior of every incoming flow can be a possible solution for detection. On the contrary, their detection techniques proposed in **[12]** require high management effort to remember every connections information, calculate suspiciousness scores and connection characteristics. The author in **[13]** proposed when the number of incomplete HTTP requests are bigger than the threshold value then the defensive measure will trigger. The threshold value of handling incomplete requests concurrently differs from server to server. Another novel metric Congestion Participation Rate **(CPR)** which can be used to identify LSDDoS flows by measuring the intention of network flows to congest the network. Random Early Detection (RED) estimates the level of congestion in the router's buffer and drops packets accordingly by maintaining an Exponentially Weighted Moving Average (EWMA) of the queue length **[14]**. But the limitation of RED algorithm is its disability to provide fairness against unresponsive flows. Overflow in OpenFlow switch (root victim) can be solved by flow table sharing with neighbor switches and Multidimensional Table Compression (MDTC) whenever an attack occurs **[15] [16]**. Neighbor switches can be flooded by that victim switch and the whole network will crash like the chain reaction. MDTC does not consider the possibility of changing the granularity of flow table size where traffic engineering and traffic monitoring mechanisms depend on the traffic flow information. Other researchers have proposed to limit the amount of traffic that a single port can send to the switch, number of packets to the controllers and limiting the number of rule installation into the switch. In **[17]** author proposed to validate the "Source IP Address" by querying the log and if the packet per second (pps) and packet bytes exceed the threshold then the flow will be removed from the switch. All previous works mentioned above are creative approaches and have the same goal of coping with LSDDoS attacks and network protection.

3 Background

3.1 Software Defined Networks

Software Defined Networks (SDN) is an architecture that aims to make networks agile and flexible. The goal of SDN is to improve network control by enabling enterprises and service providers to respond quickly according to the client's demands.

In SDN, a network engineer or administrator can shape traffic from a centralized control console without having to touch individual switches in the network. The centralized SDN controller directs the switches to deliver network services wherever they are needed, regardless of the specific connections between a server and devices. This process is a move away from traditional network architecture, in which individual network devices make traffic decisions based on their pre-configured routing tables.

Open Networking Foundation(ONF) has provided the most explicit and well-received definition of SDN as follows: "In the SDN architecture, the control and data planes are decoupled, network intelligence and state are logically centralized, and the underlying network infrastructure is abstracted from the applications"**[18]**

ONF presents a high-level architecture for SDN that is vertically split into three main functional planes including data plane, control plane and application plane as shown in Fig 3.1 **[18] [19]**.

The **data plane** consists of SDN switches (e.g. Juniper Junos MX-series, virtual switch: OpenvSwitch)**[20]** that conducts packet processing and forwarding according to the decisions made by the controller. The switches query the controller for guidance as needed and it provides the controller with information about the traffic it handles. The switch sends every packet going to the same destination along the same path and treats all the packets the exact same way.

The **control plane** consists of the centralized SDN controller that acts as the brain of SDN. This controller resides on a server and manages policies and the flow of traffic throughout the network. ONOS, OpenDaylight, Floodlight, Ryu, NOX, POX are some of the popular SDN controllers mostly used for research works.

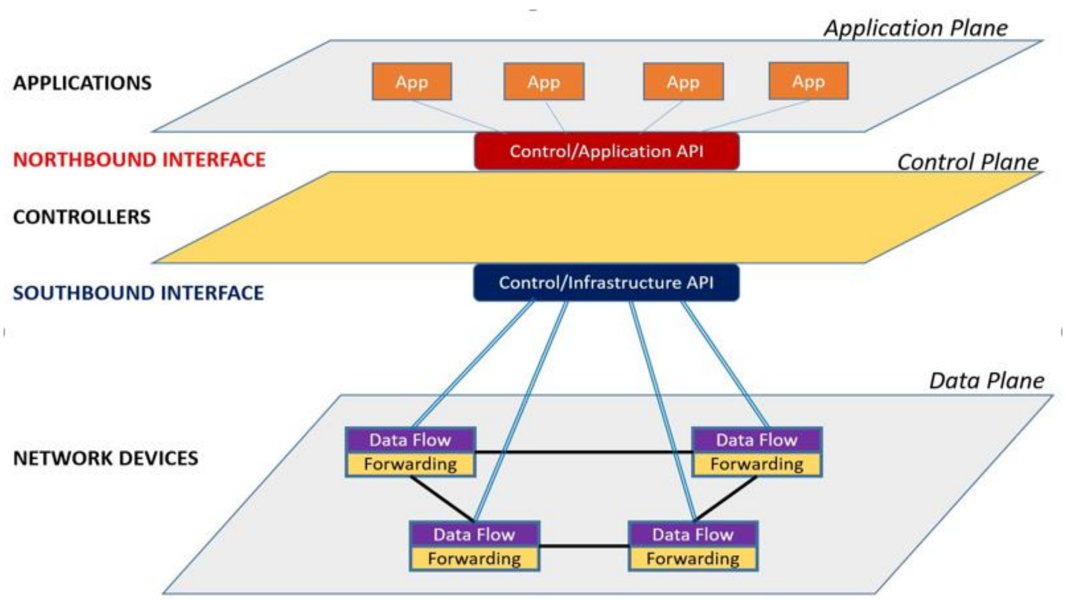


Figure 3.1: General Overview of SDNl [21]

The **application plane** contains typical network applications or functions which includes intrusion detection systems, load balancing, firewalls, and other third-party applications. The traditional network would use a specialized appliance, such as a firewall or load balancer but software-defined networks replace the appliance with an application that controller uses to manage data plane behavior. On the techno-economical point of view, operational cost is being reduced in a large amount.

These three layers communicate using respective **Northbound** and **Southbound** application programming interfaces (APIs). For example, applications talk to the controller through its northbound interface, such as REST API, while the controller and switches communicate using southbound interfaces, such as OpenFlow although other protocols also exist.

OpenFlow is, nowadays, the leading southbound protocol of SDN. OpenFlow allows a controller to define various forwarding behaviors of switches by installing related flow rules. There are two approaches for installing flow rules, i.e., proactively and reactively. In the proactive approach, flow rules are pre-installed before all the traffic comes. Pre-installed rules are LLDP, ARP, ipv4. While in reactive approach, flow rules are installed dynamically. When an OpenFlow switch receives a new packet from the hosts that does not match any installed flow rules, it generates a packet-in message to the controller for a new forwarding rule. The controller may either send packet-out messages to the switch for one-time packet processing or send flow-mod messages to install flow rules in the switches that are among the calculated routing path **[5]**.

In addition to, SDN uses an operation mode that is sometimes called adaptive or dynamic, in which a switch issues a route request to a controller for a packet that does not have a specific route. This process is separated from adaptive routing, which issues route requests through routers and algorithms based on the network topology, not through a controller.

The virtualization aspect of SDN comes into play through a virtual overlay, which is a logically separated network on top of the physical network. Users can implement end-to-end overlays to abstract the underlying network and segment network traffic. This micro-segmentation is particularly useful for service providers and operators with multi-tenant cloud environments and cloud services, as they provide a separate virtual network with specific policies for each tenant.

With SDN, an administrator can change any network switch's rules when necessary – prioritizing, deprioritizing or even blocking specific types of packets with a granular level of control and security. This is especially helpful in a cloud computing multi-tenant architecture because it enables the administrator to manage traffic loads in a flexible and more efficient manner. Essentially, this enables the administrator to use less expensive commodity switches and have more control over network traffic flow than ever before.

Other benefits of SDN are network management and end-to-end visibility. A network administrator needs to deal with only centralized controller to distribute policies to the connected switches, instead of configuring multiple individual devices. This capability is also a security advantage because the controller can monitor traffic and deploy security policies. If the controller deems traffic suspicious, for example, it can reroute or drop the packets.

Security is both a benefit and a concern with SDN technology. The centralized SDN controller presents a single point of failure and, if targeted by an attacker, can prove detrimental to the network. SDN offers new opportunities to defeat attacks in cloud computing environments. Among the security requirements of cloud computing, availability is crucial since the core function of cloud computing is to provide on-demand services of different levels. Denial of Service (DoS) attacks and Distributed Denial of Service (DDoS) flooding attacks are the main methods to destroy the availability of cloud computing. DoS or DDoS attacks are an attempt to make a machine or network resource unavailable to its intended users. DDoS attacks are sent by two or more persons, or bots, while DoS attacks are sent by one person or system. A bot is a compromised device created when a computer is penetrated by software from a malware code. Although the capabilities of SDN (e.g., software-based traffic analysis, logically centralized control, global view of the network, and dynamic updating of forwarding rules) make it easy to detect and to react DDoS attacks in cloud environments. SDN itself may be a target of some attacks, and potential DDoS vulnerabilities exist across SDN platforms. For example, an attacker can take advantages of the characteristics of SDN to launch DDoS attacks against the control layer, data plane, and application plane of SDN.

3.2 Low and Slow DoS attacks in SDN

Low and Slow attack is the newer addition to the DoS attack category. The slow attacks work very differently instead of sending requests as fast as possible, they send as slow as possible. The attacker needs to keep as many concurrent connections open as the server provides. As the attacker only needs to send a few packets per minute per connection, only little resources are necessary to render the server inaccessible. The slow body attack or slow POST attack works in similar ways. However, instead of sending the header as slow as possible the body of a POST message is split into as many packets as possible. Slow attacks are hard to detect in the network without access to the victim machine as the attackers are behaving according to specification and could easily be mistaken for clients with bad network connections **[2]**.

LSDDoS may also exhibit an **ON-OFF** nature, which comprises a succession of consecutive periods composed of an interval of inactivity (called off-time), followed by an interval of activity (called on-time). Once the LSDDoS has seized all available positions in the service queues, the attacker slows down the connections from or to the victim by exploiting the characteristics of either a specific protocol (i.e., HTTP, FTP, DNS, etc.) or the application software (i.e., PHP, SOAP, etc.). The connections are thus kept active as long as possible, by sending minimum amounts of data per time unit **[1]**.

Low and Slow attack can efficiently overflow flow tables of SDN switches by sending the minimum number of packets to the server. This attack can significantly reduce the performance of a network in disguised mode. Small-sized flow tables in OpenFlow switches will be occupied continuously by attack flows and will not leave any space for normal traffic flows because SDN controller will treat both kinds of traffic flows similarly. It is possible to overflow all the switches in the network; however, in practice attacker needs to overflow flow tables of particular switches, such as, the access switch of the target network server.

Making the server unavailable to the legitimate clients, the attacker needs crafting attack packets. The Attacker does not need to know the configurations of the whole network. Crafting of attack packets can be done in two phases which are **Probing phase** followed by the **Attacking phase [11]**.

3.2.1 Probing Phase

In this phase, an attacker infers the network configurations of flow rules with a small number of probing packets. The intention behind this scheme is that packets which do not match the flow rules in SDN switches will experience longer forwarding delays than the packets which match flow rules. This happens because switches will query the controller for the forwarding decisions and rule installation. That is why crafting probing packets and analyzing the difference in Round Trip Times (RTT) between two hosts, the attacker can accurately find out what packets will trigger new rule installation and the timeout configurations related to the flow rules.

In order to infer the configurations, particularly, match fields along with their bitmasks and timeouts which have the direct impact on plotting the attack strategies and calculating minimum attack rate which will keep the attackers flow rules alive in the flow tables. It is obligatory because different applications on the different controllers will generate flow rules with different settings**[11]**.

3.2.2 Probing Match fields

Accurately finding out the match fields in a packet header which can be used to trigger new rule installation. Attacker generates and crafts probing packets with various field values in the packet headers in the network to measure their RTTs. A probing packet can be any packet that can trigger a response packet from a destination. At first, we have sent a probing packet to a destination to trigger possible flow rule installation in switches, which ensures that a rule for the packet exists before inferring RTT. Second, we have generated a new probing packet that changes the value of one field of the previous packet to the same destination and measure the RTT (denoted by RTT_0). Then, we have sent another probing packet with the same values of header fields and measure the RTT again (denoted by RTT_1). The RTT values of the latter two packets meet the following conditions**[11]**.

$$\mathsf{RTT}_0 >> \mathsf{RTT}_1\text{; If the changed field triggers rule installation}$$
$$\mathsf{RTT}_0 \approx \mathsf{RTT}_1\text{; otherwise}$$

The First condition represents that the changed field is in the set of the match fields while another condition denotes that changed field is not in the set of the match fields. As example- The match field of IP address with a subnet mask is **"255.255.255.0"**. IP address of the first two probing packets are **"10.0.0.1"** and then following two packets are

“10.0.0.2”. If the RTTs of the last two packets are similar, then we can infer that IP address is not in the match fields of OpenFlow protocol by mistake and changing IP address will never trigger flow rule installation**[11]**. Here is the example of some basic match fields which we have used during our experiment in ONOS controller.

Protocol Name: Reactive forwarding
Component ID: org.onosproject.fwd.ReactiveForwarding

```
“matchIpv6Address”: “false”,
“matchIpv6FlowLabel”: “false”,
“ignoreIPv4Multicast”: “false”,
“matchIcmpFields”: “true”,
“flowTimeout”: “10”,
“matchTcpUdpPorts”: “true”,
“recordMetrics”: “true”,
“matchDstMacOnly”: “false”,
“matchIpv4Address”: “false”,
“matchVlanId”: “false”,
“matchIpv4Dscp”: “false”,
“ipv6Forwarding”: “false”,
“packetOutOfppTable”: “false”,
“packetOutOnly”: “false”,
“flowPriority”: “10”
```

According to OpenFlow specification 1.3 **[5]**, OpenFlow switches allow 39 match fields to install new flow rules. In Reactive forwarding protocol, match field “matchIpv4Address: false” denotes that it will not consider IPV4 address as a match field to install the flow rule. This is an example to control the network policy and depends on network administrator to orchestrate the SDN controller as per requirement.

3.2.3 Probing Timeouts

Accurately calculating timeout values of flow rules in the SDN switches is very essential to keep the flow tables overflowed before LSDDoS attack launch. If the rule matched by the packet is re-installed, then it will experience forwarding delay. Therefore, the attacker can estimate the timeout values by analyzing the elapsed time between two remarkable delays. At first, the attacker needs to probe hard timeout before idle timeout values. As a matter of fact, all timeout values can only be set to integer values and minimum value will be 1 second. For avoiding interference of idle timeout, a fixed interval between probing packets should be less than 1 second, e.g. 0.5 seconds. Thereby, the idle timeout will always be reset and will not make into effect.

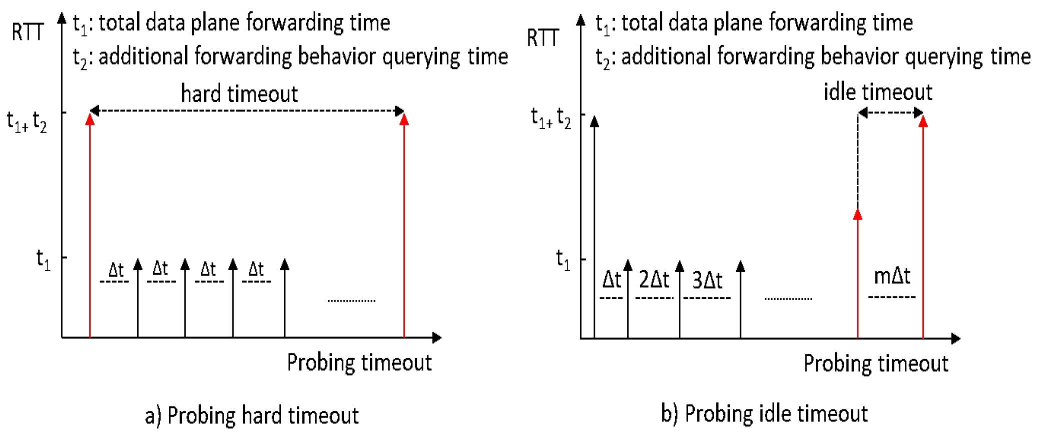

RTT
t_1: total data plane forwarding time
t_2: additional forwarding behavior querying time
hard timeout
t_1+t_2
t_1
Δt
Δt
Δt
Δt
Probing timeout
a) Probing hard timeout
RTT
t_1: total data plane forwarding time
t_2: additional forwarding behavior querying time
idle timeout
t_1+t_2
t_1
Δt
2Δt
3Δt
mΔt
Probing timeout
b) Probing idle timeout

Figure 3.2: Inferring hard and idle timeout values **[11]**

As shown in Fig 3.2 (a), to infer hard timeout values, a probing packet will be sent to trigger initial flow rule installation using the inferred match fields. Since there will be a remarkable deviation between RTTs when a new rule is installed. Periodically sending the probing packets to the network and monitoring the RTTs, the attacker can easily find out the hard timeout values as the duration since the first probing packet. It is not possible to apply same strategy to find out the idle timeout values. The reason behind that idle timeout of a flow rule will be reset once a packet matches the rule. It is required to generate and send probing packets with increasing time intervals. If a notable deviation between RTTs is found again, idle timeout values can be measured by the time interval between two successive probing packets (Fig. 3.2 (b)). In practice, there are some issues of probing timeout values as ***"Mutual Interference Between Timeouts"*** and ***"Network Jitter interference"*** **[11]**.

3.2.4 Mutual Interference Between Timeouts

Mutual interference may happen while probing timeouts because a flow rule can be removed because of either idle timeout or hard timeout or both. As an example, Flow rule is configured with hard timeout of 20 seconds and idle timeout of 15 seconds. The Time interval is increased by 1s after each round of probing idle timeout. After 20 seconds, a remarkable RTT will occur due to hard timeout. The effect of idle timeout has not taken into consideration because every time it is reset by the probing packets. 5 seconds of evaluation error of idle timeout will occur while measuring the timeouts. Hard timeout will never be less than the idle timeout values and the value has to be calculated before idle timeout **[11]**.

3.2.5 Network Jitter Interference

Another important scenario regarding RTT is the effect of *"Network Jitter interference"*. RTT values may significantly differ due to network jitter even there is no new rule installation in the switches. It is possible to eliminate the impact of network jitters using t-test method **[23]**. In t-test, a significance level α is set with a predetermined value, and a p-value is calculated according to the data, where p indicates the likelihood that the two groups of data share the same distribution. A significant difference between two groups of data is accepted if the calculated p is smaller than α. By changing the values of the same field several times, two groups of RTTs before and after the changes can be obtained. The attacker then can evaluate if two groups of RTT's are significantly different from each other by calculating their p-value. Thereby, the attacker can accurately infer if there exists new rule installation **[11]**.

3.2.6 Probing Duration

Probing idle timeout can be time-consuming if a large hard timeout is set. The total duration can be calculated as $\sum_{j=t_{idle}}^{t_{hard}} j$ where t_{idle} and t_{hard} are the idle and hard timeout respectively. Assume that, hard timeout is set to 180 s and the idle timeout value is 10s, the total time duration will be 16,425s which is approximately 4.5 hours. To effectively reduce the probing duration attacker can apply binary search in probing since it is possible to easily find out if the idle timeout is smaller or larger than the hard timeout by measuring RTTs. Interference elimination of hard timeout also has to keep in mind during binary search **[11]**.

3.2.7 Attacking Phase

Now attack can be launched in this phase according to the inferred results in the probing phase. To increase the attack effectiveness and keep it disguise, the attack will generate a minimum number of packets that can successfully overflow the tables. Calculation of minimal attack rate depends on various attack strategies which will be effective for overflow the flow tables **[11]**.

3.2.8 Crafting Attack Packets

The key insight is to ensure each packet can effectively create a unique rule installation in the switches. Using the match fields and timeout values, an attacker can successfully achieve it by carefully changing header fields values of each packet. Thereby, the minimal number of packets to overflow flow tables of a switch is equal to the flow table size. Attack packets usually do not carry real payload. Attacker can generate minimum size of Ethernet packets which is 64 bytes **[20]**. Approximately 113 KB traffic can successfully overflow a switch with 1800 rules. Hence, the volume of the total attack packet is small, the attacker can also use multiple match fields with different values in the attack packets to disguise the attack packets as benign packets. As an example, if a flow rule is configured with IP source address, IP destination address and TCP source port, the attacker can change the IP source address in few packets while change TCP source port in other packets. In addition to, generation of different payloads in attack packets and random packet lengths can also be generated during the attack **[11]**.

3.2.9 Minimal Attack rate calculation

An Attacker can generate different attack packets according to different time settings. Assume, x and y are the integers and x > y. There are four categories according to hard timeout and idle timeout:

Table 3.1: Different timeout Settings**[11]**

Category	**Hard**	**Idle**	**Detail**
1	0	0	Flow rule exists permanently
2	x	0	Flow rule will be removed after x second
3	0	y	Rule will be removed after 'y'-if no packets match the rule
4	x	y	Rule will be removed either x or y sec

Among the above, category (3) and (4) time settings are mostly used in default settings of different controllers (Table 3.2) and rests are rarely used. If the settings in category (1) are implemented, a significant amount of resources in the controller are required to actively monitor all flow rules such that flow rules will be removed when there are no matching packets. While the settings in category (2) cannot ensure that flow rules can be removed in time if the network does not generate any packets matching the rules, resulting in the waste of the scarce flow table resources **[11]**. Following table is an example of different timeout values of different controllers.

Table 3.2: Default timeout values in different Controllers **[11]**

Controller	**Floodlight**	**Maestro**	**NOX**	**ONOS**	**OpenDaylight**	**POX**
Hard timeout	0	180s	0	0	600s	30s
Idle timeout	5s	30s	5s	10s	300s	10s

3.2.10 Attack Strategy with category (3) and (4) settings

According to category (3) settings, within an idle timeout period, an attacker requires to fill in the flow table and ensure consumption's of entire flow table after the idle timeout expires. The attacker will try to achieve it by generating and distribute them evenly within each idle timeout period. The packets will trigger a new rule installation and the number of rules in the flow table will gradually increase. Keeping the flow rules alive in the flow table continuously, idle timeout of each flow rule needs to be periodically refreshed within each idle timeout interval. The average rate of sending attack packets within an idle timeout period can be calculated by:

$$\bar{v} = \frac{\sum_{i=0}^{C-1} L_i}{t_{idle}}$$

Here, $\bar{v}$ is the average packet rate, C represents the maximum capacity of a flow table, L_i denotes the length of the i^{th} packets with an idle timeout period. In order to fully consume the maximum capacity of the flow table, an attacker needs to generate at least C packets. Attack rate is less than $\bar{v}$ cannot fully consume the table and keep the table full over time because flow rules will be expired after the idle timeout. For example: if the flow table capacity is 1800 flow rules, the idle time out is 20 seconds and the packet size is 64 bytes, then the minimal attack rate to overflow flow tables is only 64 Kbps. This type of low attack rate ensures that malicious rules will not expire and the attack traffic can be disguised as legitimate traffic.

Category (4) setting ensures that flow rules will be removed either hard timeout or idle timeout expires. Sending packets to overflow flow tables gradually within an idle timeout and refreshing the flow rules, the attacker needs to reinstall the flow rule once it is removed due to hard timeout. An attacker can easily achieve it as timeout settings have been known from the probing phase**[11]**.

3.3 Self Organizing Maps

Self Organizing Maps (SOM) **[22]** is an artificial neural network which transforms a given n-dimensional pattern of data into a 1- or 2-dimensional map or grid. This transformation process is done following a topological ordering, where patterns of data (synaptic or vector weights) with similar statistical features are gathered in regions close to each other in the grid. This learning process can be classified as competitive based because neurons compete against each other to be placed at the output layer of the neuron network, but only one wins, as illustrated by Fig 3.3. It is also called unsupervised algorithm because the neuron network learns only with entry patterns, re-organizing itself after the first trained data and adjusting its weights as new data arrive. A detailed description of the complete SOM algorithm is presented in **[22]**. In what follows, we provide a summary of the main steps of how the SOM's learning process works.

1) **Initialization:** at the beginning of the process all neuron vectors have their synaptic weights randomly generated. Such vectors must have the same dimension of the entry pattern space.
2) **Sampling:** a single sample x is chosen from the entry pattern space and fed to the neuron

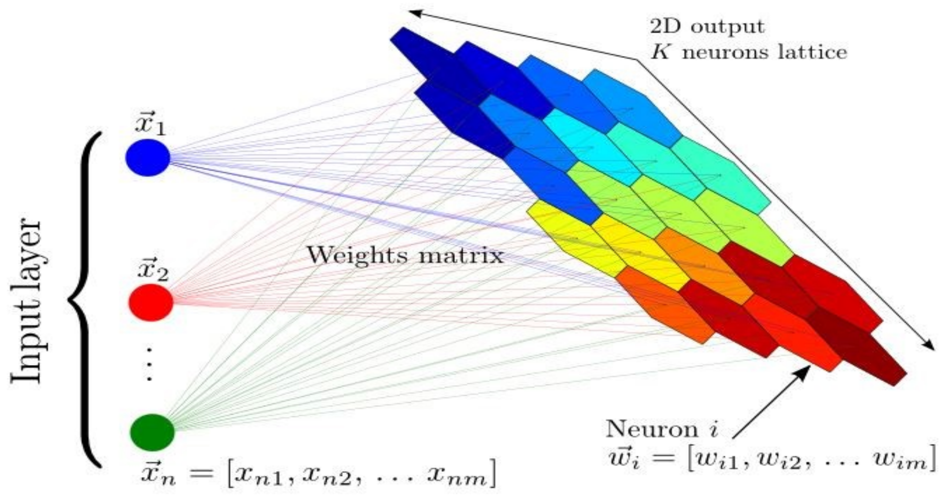

2D output
K neurons lattice
$\vec{x}_1$
Input layer
$\vec{x}_2$
Weights matrix
$\vec{x}_n = [x_{n1}, x_{n2}, \dots x_{nm}]$
Neuron i
$\vec{w}_i = [w_{i1}, w_{i2}, \dots w_{im}]$

Figure 3.3: Kohonen's SOM map **[22]**

grid.
3) **Competition:** based on the minimum Euclidean distance criterion the winning neuron i(x) is found as follows: $i(x) = argmin(\| x{=}W_j \|)$, j=1, 2, ..., where i is the number of neurons in the grid.
4) **Synaptic adaptation:** After finding the winning neuron, all synaptic weights of each neuron vectors are adjusted: $W_j(t+1) = W_j(t) + \eta(t)\theta_j(t)(x(t) - W_j(t)$ where t represents the current instant, $\eta(t)$ is the learning rate which gradually decreases with time t, and $\theta_j(t)$ is the neighborhood function which determines the grade of learning of a neuron j according to its relative distance to the winning neuron.
5) Repeat steps 2 to 4 until no significant change happens in the topological map.

4 Source-based Intelligent Detection of Low and Slow DDoS Attacks in Software Defined Networks

4.1 Feature Selection

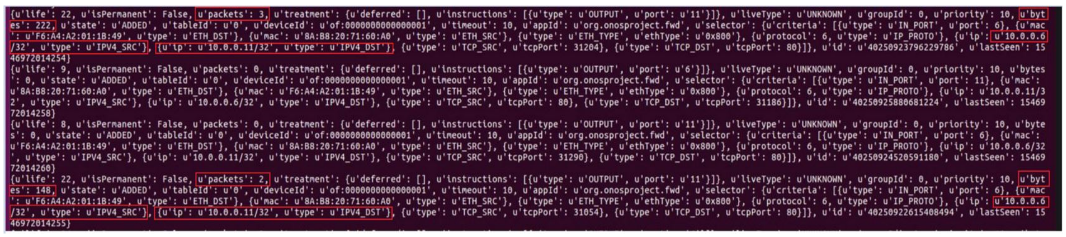
LSDDoS traffic has some basic parameters as low packet count, the creation of flow rules using similar source IP address, source port and small packet size. We are proposing these 4-tuples to accurately identify the attack and the attack source. Our proposed parameters are : 1) **Average Packet per flow (APf)**, 2) **Average Packet Size per flow (APSf)**, 3) **Packet Change Ratio (PCR)**, 4) **Flow Change Ratio (FCR)**. These parameters are the feature for our ML (SOM) algorithm as well.

1) Average Packet per flow (APf): Though LSDDoS works very different than usual DDoS attack, instead of sending many packets at once as LSDDoS attack sends a very small number of packets. Then sending these packets at a very low rate (e.g. 0.5 Hz; 10s-30s between packets). Attacker needs to keep as many concurrent connections open as the server provides. For example, attacker sends 3 packets per flow (Fig. 4.2) whereas the legitimate user sends more than 300 packets per flow (Fig. 4.1). Equation 4.1 is then used to compute the average packet per flow for that IP address which is sending requests to the server.

$$APF \ = \frac{P_2 - P_1}{num - flows} \tag{4.1}$$

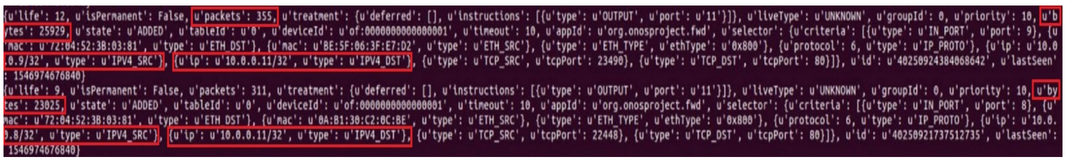
where P_2 is the latest packet counts, P_1 is the previous packet counts and num-flows is the number of flows created by that IP.

2) Average Packet Size per flow (APSf): Though HTTP get requests are split in many packets, packet size is very smaller than the normal packets. Attacker meets the minimum



```
{u'life': 22, u'isPermanent': False, u'packets': 3, u'treatment': {u'deferred': [], u'instructions': [{u'type': u'OUTPUT', u'port': u'11'}]}, u'liveType': u'UNKNOWN', u'groupId': 0, u'priority': 10, u'byt
es': 222, u'state': u'ADDED', u'tableId': u'0', u'deviceId': u'of:0000000000000001', u'timeout': 10, u'appId': u'org.onosproject.fwd', u'selector': {u'criteria': [{u'type': u'IN_PORT', u'port': 6}, {u'mac
': u'F6:A4:A2:01:1B:49', u'type': u'ETH_DST'}, {u'mac': u'8A:B8:20:71:60:A0', u'type': u'ETH_SRC'}, {u'type': u'ETH_TYPE', u'ethType': u'0x800'}, {u'protocol': 6, u'type': u'IP_PROTO'}, {u'ip': u'10.0.0.6
/32', u'type': u'IPV4_SRC'}, {u'ip': u'10.0.0.11/32', u'type': u'IPV4_DST'}, {u'type': u'TCP_SRC', u'tcpPort': 31204}, {u'type': u'TCP_DST', u'tcpPort': 80}]}, u'id': u'40250923796229786', u'lastSeen': 15
46972014254}
{u'life': 9, u'isPermanent': False, u'packets': 0, u'treatment': {u'deferred': [], u'instructions': [{u'type': u'OUTPUT', u'port': u'6'}]}, u'liveType': u'UNKNOWN', u'groupId': 0, u'priority': 10, u'bytes
': 0, u'state': u'ADDED', u'tableId': u'0', u'deviceId': u'of:0000000000000001', u'timeout': 10, u'appId': u'org.onosproject.fwd', u'selector': {u'criteria': [{u'type': u'IN_PORT', u'port': 11}, {u'mac':
u'8A:B8:20:71:60:A0', u'type': u'ETH_DST'}, {u'mac': u'F6:A4:A2:01:1B:49', u'type': u'ETH_SRC'}, {u'type': u'ETH_TYPE', u'ethType': u'0x800'}, {u'protocol': 6, u'type': u'IP_PROTO'}, {u'ip': u'10.0.0.11/3
2', u'type': u'IPV4_SRC'}, {u'ip': u'10.0.0.6/32', u'type': u'IPV4_DST'}, {u'type': u'TCP_SRC', u'tcpPort': 80}, {u'type': u'TCP_DST', u'tcpPort': 31186}]}, u'id': u'40250925880681224', u'lastSeen': 15469
72014258}
{u'life': 8, u'isPermanent': False, u'packets': 0, u'treatment': {u'deferred': [], u'instructions': [{u'type': u'OUTPUT', u'port': u'11'}]}, u'liveType': u'UNKNOWN', u'groupId': 0, u'priority': 10, u'byte
s': 0, u'state': u'ADDED', u'tableId': u'0', u'deviceId': u'of:0000000000000001', u'timeout': 10, u'appId': u'org.onosproject.fwd', u'selector': {u'criteria': [{u'type': u'IN_PORT', u'port': 6}, {u'mac':
u'F6:A4:A2:01:1B:49', u'type': u'ETH_DST'}, {u'mac': u'8A:B8:20:71:60:A0', u'type': u'ETH_SRC'}, {u'type': u'ETH_TYPE', u'ethType': u'0x800'}, {u'protocol': 6, u'type': u'IP_PROTO'}, {u'ip': u'10.0.0.6/32
', u'type': u'IPV4_SRC'}, {u'ip': u'10.0.0.11/32', u'type': u'IPV4_DST'}, {u'type': u'TCP_SRC', u'tcpPort': 31290}, {u'type': u'TCP_DST', u'tcpPort': 80}]}, u'id': u'40250924520591180', u'lastSeen': 15469
72014260}
{u'life': 22, u'isPermanent': False, u'packets': 2, u'treatment': {u'deferred': [], u'instructions': [{u'type': u'OUTPUT', u'port': u'11'}]}, u'liveType': u'UNKNOWN', u'groupId': 0, u'priority': 10, u'byt
es': 148, u'state': u'ADDED', u'tableId': u'0', u'deviceId': u'of:0000000000000001', u'timeout': 10, u'appId': u'org.onosproject.fwd', u'selector': {u'criteria': [{u'type': u'IN_PORT', u'port': 6}, {u'mac
': u'F6:A4:A2:01:1B:49', u'type': u'ETH_DST'}, {u'mac': u'8A:B8:20:71:60:A0', u'type': u'ETH_SRC'}, {u'type': u'ETH_TYPE', u'ethType': u'0x800'}, {u'protocol': 6, u'type': u'IP_PROTO'}, {u'ip': u'10.0.0.6
/32', u'type': u'IPV4_SRC'}, {u'ip': u'10.0.0.11/32', u'type': u'IPV4_DST'}, {u'type': u'TCP_SRC', u'tcpPort': 31054}, {u'type': u'TCP_DST', u'tcpPort': 80}]}, u'id': u'40250922615408494', u'lastSeen': 15
46972014255}
```

Figure 4.1: Normal Flow Statistics



```
{u'life': 12, u'isPermanent': False, u'packets': 355, u'treatment': {u'deferred': [], u'instructions': [{u'type': u'OUTPUT', u'port': u'11'}]}, u'liveType': u'UNKNOWN', u'groupId': 0, u'priority': 10, u'b
ytes': 25929, u'state': u'ADDED', u'tableId': u'0', u'deviceId': u'of:0000000000000001', u'timeout': 10, u'appId': u'org.onosproject.fwd', u'selector': {u'criteria': [{u'type': u'IN_PORT', u'port': 9}, {u
'mac': u'72:04:52:3B:03:81', u'type': u'ETH_DST'}, {u'mac': u'BE:5F:06:3F:E7:D2', u'type': u'ETH_SRC'}, {u'type': u'ETH_TYPE', u'ethType': u'0x800'}, {u'protocol': 6, u'type': u'IP_PROTO'}, {u'ip': u'10.0
.0.9/32', u'type': u'IPV4_SRC'}, {u'ip': u'10.0.0.11/32', u'type': u'IPV4_DST'}, {u'type': u'TCP_SRC', u'tcpPort': 23490}, {u'type': u'TCP_DST', u'tcpPort': 80}]}, u'id': u'40250924384068642', u'lastSeen'
: 1546974676840}
{u'life': 9, u'isPermanent': False, u'packets': 311, u'treatment': {u'deferred': [], u'instructions': [{u'type': u'OUTPUT', u'port': u'11'}]}, u'liveType': u'UNKNOWN', u'groupId': 0, u'priority': 10, u'by
tes': 23025, u'state': u'ADDED', u'tableId': u'0', u'deviceId': u'of:0000000000000001', u'timeout': 10, u'appId': u'org.onosproject.fwd', u'selector': {u'criteria': [{u'type': u'IN_PORT', u'port': 8}, {u'
mac': u'72:04:52:3B:03:81', u'type': u'ETH_DST'}, {u'mac': u'0A:B1:30:C2:0C:BE', u'type': u'ETH_SRC'}, {u'type': u'ETH_TYPE', u'ethType': u'0x800'}, {u'protocol': 6, u'type': u'IP_PROTO'}, {u'ip': u'10.0.
0.8/32', u'type': u'IPV4_SRC'}, {u'ip': u'10.0.0.11/32', u'type': u'IPV4_DST'}, {u'type': u'TCP_SRC', u'tcpPort': 22448}, {u'type': u'TCP_DST', u'tcpPort': 80}]}, u'id': u'40250921737512735', u'lastSeen':
 1546974676840}
```

Figure 4.2: Attack Flow Statistics

requirements of packet size which is 64 bytes. Packet size for each packet in attack traffic flow is around 64-70 bytes whereas packet size in normal traffic is usually more than 350 bytes (we have used GNU wget to download some jpeg files from the server that is why packet size seems similar to attacker in fig. 4.1. In practice packet size of a normal users is more than 350 bytes). Equation 4.2 is then used to compute the average packet size per flow:

$$APSF = \frac{B_2 - B_1}{num - flows} \tag{4.2}$$

where B_2 is the latest packet size, B_1 is the previous packet size and num-flows is the number of flows.

3) Packet Change Ratio (PCR): LSDDoS attacker's intention is to keep all the connections alive in the SDN switch and open new connections continuously. For opening new connections until the switch is overflowed, the attacker will increase the number of sending packets within the idle-timeout period whereas normal user sends packets for a certain period of time. In addition to, we can easily conclude that packet change ratio for attacker will always be positive and for the normal user will be negative and sometimes positive (for a new user who is connected for the first time in a network). Equation 4.3 is used for calculating packet change ratio:

$$PCR = \frac{P_2 - P_1}{P_2 \text{ or } P_1} \tag{4.3}$$

where P_2 is the latest packet counts and P_1 is the previous packet counts. If packet count decreases then P_1 will be used or for increment P_2 will be used to calculate the ratio.

4) Flow Change Ratio (FCR): To keep all the connections alive in SDN switch, attacker has to send packets from the same IP source and port periodically otherwise the flow will be expired after idle-timeout and attacker will always try to increase the number of flow rules in the SDN switch to overflow. So that, that flow change ratio for attacker will always be positive and normal user will be negative. On the other hand, the normal user's flow will be dead after idle-timeout or hard-timeout. Equation 4.4 is used for calculating flow change ratio:

$$FCR = \frac{f_2 - f_1}{f_2 \text{ or } f_1} \tag{4.4}$$

where f_2 is the latest flow counts and f_1 is the previous flow counts. If packet count decreases then f_1 will be used or for increment f_2 will be used to calculate the ratio.

4.2 Architecture

Fig. 4.3 provides an overview of the extended SDN control plane. It consists of default SDN control plane with our novel LSDDoS detection engine in application plane. Detailed description of the functionalities of the detection engine is given below.

1) Statistics Collector: The ***Statistics Collector*** module is responsible for periodically requesting flow entries from flow tables of OpenFlow (OF) switches from SDN controller via RESTful API **[25], [26]**. Such requisitions, as well as their corresponding answers, are transmitted through a Secure Channel, i.e. a channel that is isolated from hosts connected to the switches.

2) Analyzer: The ***Analyzer*** module receives collected flows from monitor threads of SDN switches via the Statistics collector and check the SDN switches for source IP addresses which are intended to access the server. Then the module extracts features that are important to LSDDoS flooding attack detection and gathers them in 4-tuples to be passed through the classifier. Section 4.1 gives a more detailed explanation of this 4-tuple. Every 4-tuple is associated with the switch it belongs by a switch ID, which can be easily obtained by the SDN controller.

3) Classifier (SOM): The ***Classifier*** module analyzes whether a given 4-tuple corresponds to a LSDDoS flooding attack or to a legitimate traffic. This classification can be made by any statistical or learning method. In our work, we used SOM as the classification method. The reason behind choosing SOM as our classification algorithm is it is capable of dealing real-time data. We have tried SVM algorithm which is not efficient enough to classify the attack source.

The SOM algorithm in the detection engine works as follows. In general, we do not have a linearly separable data set for training $D = (x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)$, where $y_i \epsilon (0, 9)$. In this work, x_i represents the 4-tuple (APf, APSf, PCR, FCR). The value of y_i represents the number of clusters in SOM. Before SOM is able to classify 4-tuple collected by the extractor module, it needs to be trained with a sufficiently large set of 4-tuple samples collected during attack traffic as well as normal traffic. In this way, SOM is able to create a topological map where different regions represent each kind of traffic. Then, whenever the trained SOM is stimulated with a 4-tuple extracted from collected flow entries of OF switch, it will be able to classify it either as normal traffic or attack traffic flow. The amount

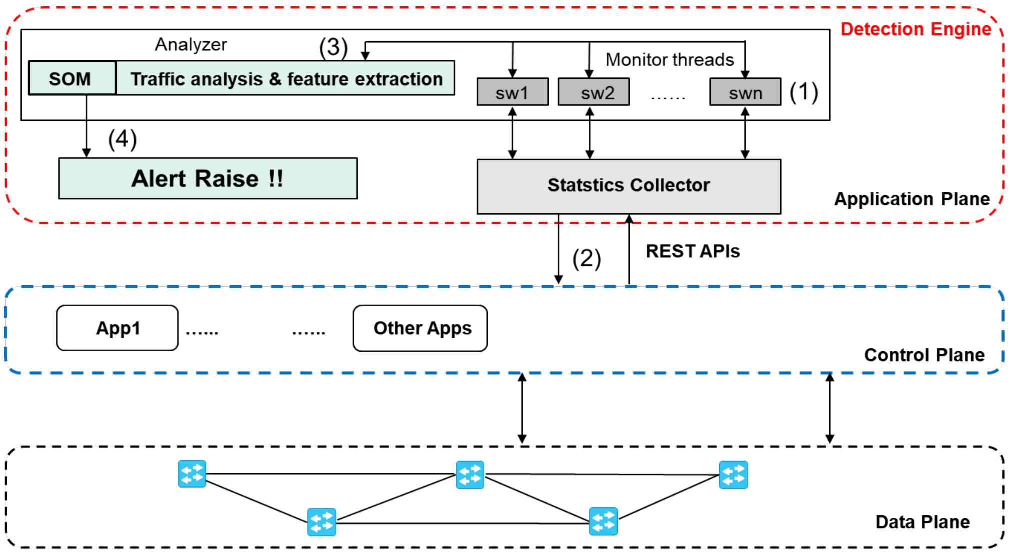


Figure 4.3: LSDDoS Detection Loop Operation

of 4-tuples used for the training phase is presented in Chapter V. Whenever, 4-tuple is classified by SOM as an attack, an alert and the attacking source IP will be immediately sent to the network administrator. Our implementation of SOM consists of a 10 × 10 matrix of neurons and to compute the topological neighborhood Euclidean distance function is used. The value belongs to cluster 5 denotes the attack traffic and rest of the clusters are defined as benign traffic.

4.3 Operational Workflow

An OF switch needs to be authenticated by the SDN controller which generates an ID for the switch. Only after authentication is complete all hosts are connected to the switch allowed to exchange packets. Every packet which arrives in an OF switch has its header matched against flow entries of the switch's Flow Table. In case some entries successfully match the packet's header, it's statistics are updated (for instance, the number of packet size and packets are increased). Otherwise, if no flow entry (in the switch's Flow Table) matches the packet's header, then the entry is forwarded to the controller. In its turn, the controller may add, according to the defined policy, a new flow entry to the flow table of every switch as required for policy enforcement. Thus, the traffic generated by all hosts connected to a given OF switch will populate the flow table of said switch. The collection of flow entries from an OF switch is performed at predetermined time intervals by the controller. From this collection, proposed features are extracted to classify traffic as normal or as an attack.

As the Statistics collector gathers samples from the OF switch authenticated by the controller, the switch ID is used to help the classifier module to pinpoint in which OF switch LSDDoS flooding attacks are detected (In our case, we have used only 1 switch for test purpose). The definition of the time interval to collect flow entries is of great importance. If the collection is made at infrequent time intervals, then there will be a delay to detect an attack and consequently a reduction of the time available for the possible mitigation. On the other hand, if the time interval for collection is too short, there will be an increase of packets requesting flows which will lead to an increament in the overhead of our detection mechanism. By default, every OpenFlow switch registered at the SDN controller is automatically added into the detection loop. But the network administrator can also restrict sample collection to those switches that are most relevant. As the controller manages the information of network switches in a centralized way, we are able to monitor all selected switches and analyze their traffic of that LSDDoS attack. **Algorithm 1** represents the basic workflow of our LSDDoS module.

Algorithm 1: LSDDoS Detection Module

```
begin
SOM initialization;
loop
while True do
    response ← flows_from_SW1
    if [response][flows][ip][IPV4_DST] = server IP then
        if source_list.count.(IPV4_SRC)==0 then
            source_list.append(IPV4_SRC);
            count_ob += 1
        end
        x = {APf,APSf,PCR,FCR} ; {Calculation of proposed parameters}
        y = SOM(x); {feed x into SOM}
        index= MAP [y] ; {SOM will return which cluster y belongs to}
        if count_ob == n ; {n=observation number (5,7,10)} then
            if index == 5 then
                Black_list.append.(IPV4_SRC)
                print IPV4_SRC ; {print attack source}
            else
                continue
            end
        end
    else
        time.sleep()
    end
end
```

Initially, the Statistics Collector sends a request to the SDN controller to ask for network topology information. Then, it launches a monitor thread for each connected SDN switch (step (1) in Fig. 4.3). Next, the Statistics Collector gets traffic flow statistics from the connected SDN switches (step (2) in Fig. 4.3). In regular time intervals (observation period) - for each SDN switch - a monitor thread counts packets, packet size, average packets per flow, average packet size, packet number change and the flow number changes in order to provide the tuple (APf, APSf, PCR, FCR) to the SOM engine within the Analyzer (step (3) in Fig. 4.3). Meanwhile, the Analyzer activates the SOM engine and performs the training phase using a pre-prepared training data set see (Chapter 5.2). The Analyzer then conducts traffic analysis and fed the parameters to the SOM engine and then the detection engine will raise alert whenever necessary (step (4) in Fig. 4.3).

5 Experiments

This section gives readers an elaborated implementation of the flow-based mechanism in Software-Defined Networking to detect LSDDoS attacks and attack source. Fig. 5.1 shows our testing topology which consists of the SDN controller (ONOS), an OpenvSwitch, a web server, normal hosts and attacker hosts. For generating topology, MaxiNet framework **[24]** has applied to emulate SDN network consists of only 1 SDN switch (OpenvSwitch), 10 enterprise hosts and a web server (see Fig. 5.1). This emulated network runs within one Linux PC along with the hosts and is controlled by a remote SDN ONOS controller **[25]** running on another Linux PC. Hosts are running within LINUX docker-containers using Ubuntu images and the server is running within a LINUX docker-container using Apache Web server images. 6 of the 10 enterprise hosts contain the LSDDoS attack tool along with some other applications. These docker are uploaded into the docker hub repository. For a convenient configuration, we place both the ONOS SDN controller and the LSDDoS detection engine on the same Linux PC (see Fig. 5.2).

Our experiments are conducted using both normal and attack datasets to train SOM map. Besides, an LSDDoS attack tool is used to generate abnormal flows, while normal flows are generated by using GNU wget. It is a linux tool used for downloading images from the server. Continuous data flow is kept between hosts and server.

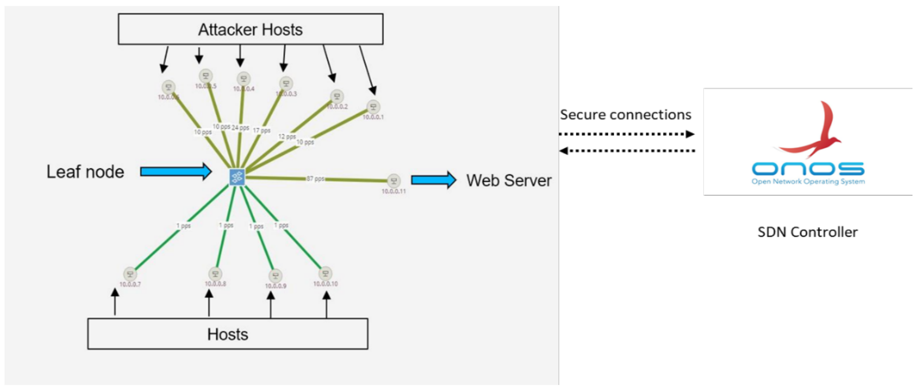


Figure 5.1: Network Topology

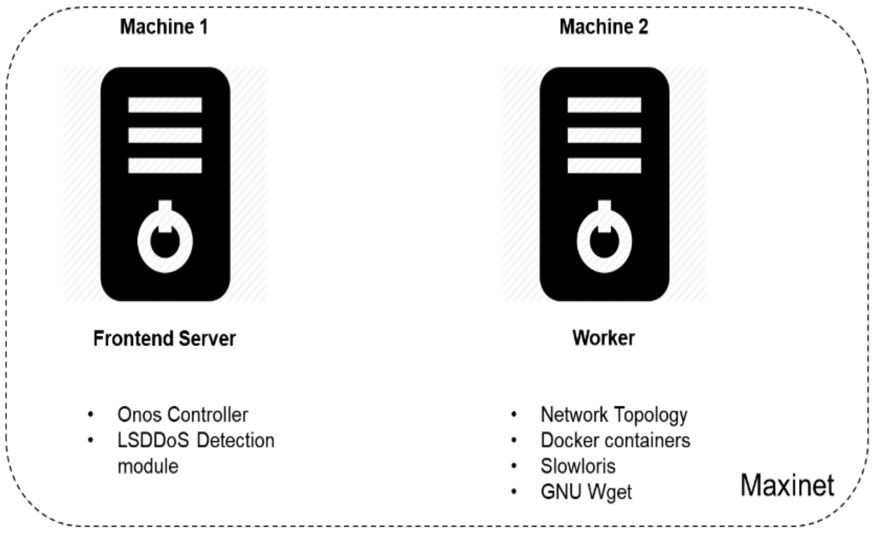


Figure 5.2: Testbed Setup

5.1 LSDDoS Attack Tool

In our experiment, we have executed LSDDoS tool named "Slowloris" **[27]**. This tool is able to simulate LSDDoS attack using botnets and can generate TCP packets to target network from defined botnets. This tool is very easy to modify as per the network configurations inferred in probing phase. Attack can be generated using following command:

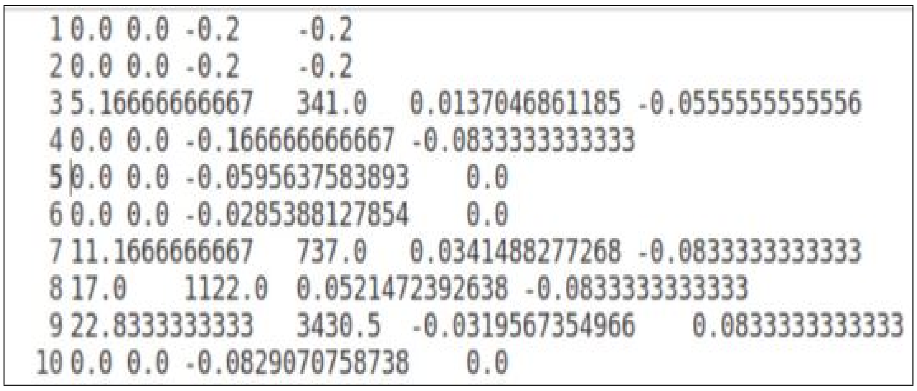
```
$ perl slowloris.pl —dns —port —timeout —num
```

In this command 4 variables are changed to generate random packet.

- dns - Here target IP address/server name to where the packet is sent.
- port - Here which port number of the server to attack. i.e FTP:21,HTTP:80, SMTP:25
- timeout - Here to mention the time interval of sending attack packets.
- num - Here number of connections to be opened in the server.

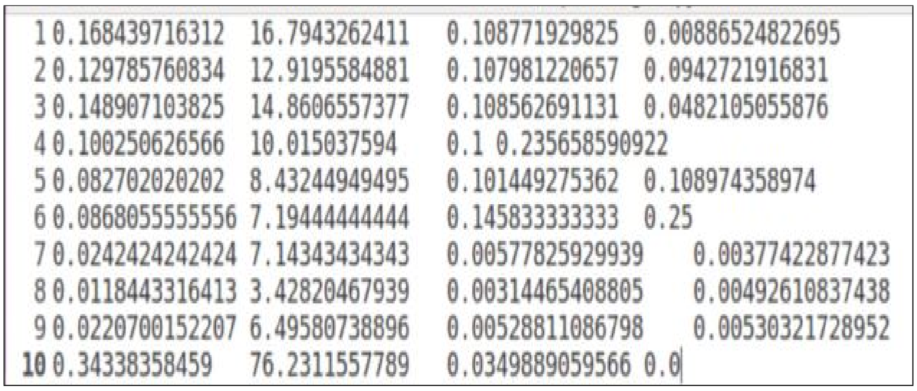
The default idle-timeout of ONOS SDN controller is 10s and 500s for APACHE Web server. This idle-timeout is required for keeping the flow rules alive successfully in the SDN switch. We have used only 1 switch for the test purpose. We will use this detection module in a large scale of the network in future. 6 hosts generate 400 TCP connections with time interval of 5. Attacker will fix the port number for the connections because attacker has to send packets from same TCP source port otherwise flow rules will be removed from flow table. The following command is an example for Linux to open particular TCP source port range:

```
echo 15000 15500 > /proc/sys/net/ipv4/ip_local_port_range
```



```
0.0 0.0 -0.2    -0.2
0.0 0.0 -0.2    -0.2
5.16666666667   341.0   0.0137046861185 -0.0555555555556
0.0 0.0 -0.166666666667 -0.0833333333333
0.0 0.0 -0.0595637583893    0.0
0.0 0.0 -0.0285388127854    0.0
11.1666666667   737.0   0.0341488277268 -0.0833333333333
17.0    1122.0  0.0521472392638 -0.0833333333333
22.8333333333   3430.5  -0.0319567354966    0.0833333333333
0.0 0.0 -0.0829070758738    0.0
```

Figure 5.3: Dataset sample from Normal users



```
0.168439716312  16.7943262411   0.108771929825  0.00886524822695
0.129785760834  12.9195584881   0.107981220657  0.0942721916831
0.148907103825  14.8606557377   0.108562691131  0.0482105055876
0.100250626566  10.015037594    0.1 0.235658590922
0.082702020202  8.43244949495   0.101449275362  0.108974358974
0.0868055555556 7.19444444444   0.145833333333  0.25
0.0242424242424 7.14343434343   0.00577825929939    0.00377422877423
0.0118443316413 3.42820467939   0.00314465408805    0.00492610837438
0.0220700152207 6.49580738896   0.00528811086798    0.00530321728952
0.34338358459   76.2311557789   0.0349889059566 0.0
```

Figure 5.4: Dataset sample from Attackers

By default 00000 to 65535 ports are open. We have defined 500 ports in every attack docker-containers as our intention is to overflow the flow table of the switches successfully.

5.2 Datasets

In the beginning, for training SOM we have generated legitimate traffic from enterprise hosts and taken more than 2000 data of our proposed parameters. After generating normal user parameters, we have initiated LSDDoS attack from the enterprise hosts via Slowloris attack tool. Then using Statistics collector and Analyzer module we have collected more than 2000 data of our proposed parameters from the attacker behaviors. A sample from the normal flow dataset and the attack flow dataset is shown in Fig. 5.3 and Fig. 5.4.

All of the above datasets are arranged as (APf, APSf, PCR, FCR) respectively. These tuple samples are then used for training the SOM and it has created 10 clusters to group the similar data. Whenever a flow rule is created, Analyzer checks the source IP address which is intended to access the server. If the flow entry contains Server IP address as destination address then Analyzer extracts the tuples from that flow entries of that source IP address and fed into the SOM to distinguish between normal and malicious users.

5.3 Training and Testing Process

Before launching tests, we carefully prepare datasets for training the SOM map as presented in Table 5.1. We train SOM with 400 samples extracted from both normal traffic and attack traffic datasets.

The testing procedure starts with two scenarios: the system only contains normal flows and the system is under TCP flooding from 6 attackers. Rest of the hosts are sending usual TCP traffic to the web server. In the first scenario, wget tool has installed in legitimate users who access to the server and download image file. It generates up to more than 1000 flows and done by 4 enterprise hosts. Meanwhile, in the attack scenario, the Slowloris tool has installed in illegal users launch and each attack hosts can produce more than approximately 500 abnormal flows to our web server while normal TCP traffic flow goes on. We tested our detection module for 300 tests in the emulated environment based on several observations (e.g. observation number = 5,7,10). In each observation, analyzer extracts feature from the flows and classify them. After a particular observation number SOM will map the features and classify the hosts either normal hosts or attacker. We tested our module in an emulated environment when 4 hosts are sending normal traffic flows and 1 attacker attacks the server connected switch by Slowloris tool.

Table 5.1: Number of Training samples

	Normal Traffic Phase	**Attack Traffic Phase**
OF Switch 1	400	400

6 Results and Performance Evaluation

In this section detailed experiment results are demonstrated along with comprehensive analyses are also provided hereafter.

Table 6.1: Experiment Results

	Observation Number	**TP(%)**	**TN(%)**	**FP(%)**	**FP(%)**
	5	94	91	9	6
SOM	7	98	96	5	2
	10	89	97	3	11

6.1 Detection Rate, Accuracy, Precision and FAR enhancement

We have launched 100 attacks from a single hosts and tested our detection module every time. After the completion of experiment, we have computed 4 parameters: **True Positive** (TP) is the probability of abnormal flows which are classified as illegal flows. **True Negative** (TN) shows the probability of legal flows that are trusted as normal flows. **False Positive** (FP) reflects the probability of normal flows that are accepted as abnormal flows and **False Negative** (FN) reveals the probability of attack flows that are recognized as legal flows. Table 6.1 depicts values of all parameters.

From the results mentioned in Table 6.1 we have calculated four most important criteria, **Detection Rate (DR), Accuracy, False Alarm Rate (FAR), Precision** to evaluate the performance of our detection module. Detection rate is the ratio of correctly identified attack over the total amount of attacks are happened in the networks. On the other hand, precision represents the ratio of correctly identified attacks over the total amount of identification of attacks. False Alarm Rate represents the ratio of incorrectly identified attack over the total amount of incorrect identification whereas, Accuracy means how correct our module is. Above mentioned criteria computation formula is mentioned in Table 6.2.

It can be seen from the Fig 6.1 and Table 6.3 that when observation number is 10 our detection module has the best performance. As we increase the observation number FP decreases and TP increases. The more we increase the observation number the more accurate our detection module is. Though detection time increases which is essential to detect the attack source accurately rather than detecting a legitimate user as an attacker. From

Table 6.2: Formula of Performance Indices

Detection Rate	Accuracy	Precision	FAR
$\frac{TP}{TP+FN}$	$\frac{TP+TN}{TP+FP+TN+FN}$	$\frac{TP}{TP+FP}$	$\frac{FP}{TN+FP}$

Table 6.3: Detection Criteria Results

Observation Number	Detection Rate (%)	Accuracy (%)	Precision (%)	FAR (%)
5	94	95.21	91.26	9.57
7	98	97.46	95.14	5.05
10	89	98.31	96.74	3.37

Fig. 6.2, we can see that most of the detection is within the region of 30s to 60s.It represents that our detection module can detect the attack source within a minute or less which is very important. Regarding Accuracy and Precision, we have got the best performance which is 98.31% and 96.73% respectively for the observation number 10. Detection rate is much lower in observation number 10 than other observation numbers because we have tested our detection module for 100 times for each observation number. 89% detection rate is still high but at least 1000 tests are required to get better performance for every observation number. In respect of the False Alarm rate, observation 10 presents best performance for keeping False Alarm Rate(FAR) which is 3.37% while the rate of incorrect alarm generated by other observation numbers are around 9.6% and 5.0%. In summary, our proposed detection module has very low detection time, false alarm rate but high detection rate.

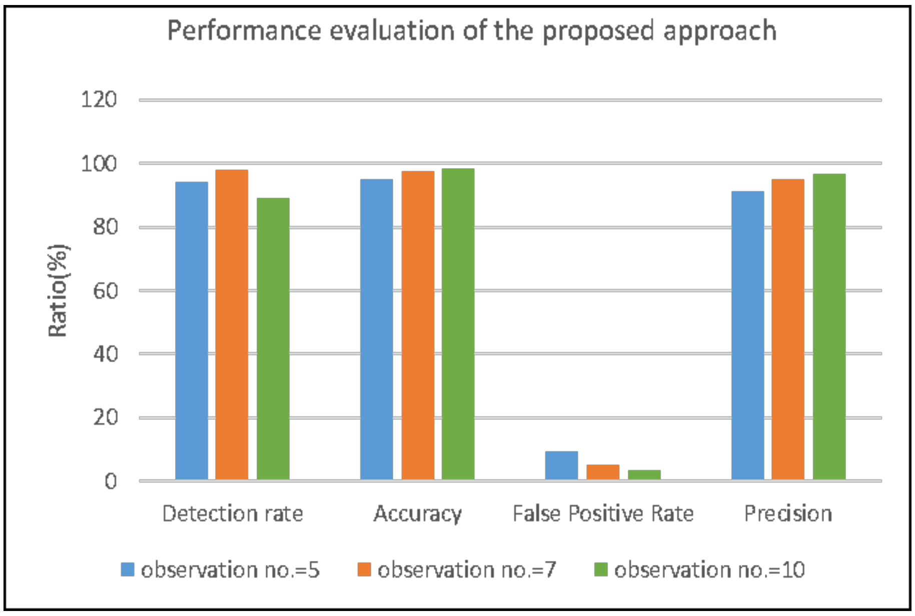


Figure 6.1: Performance Evaluation for the proposed approach

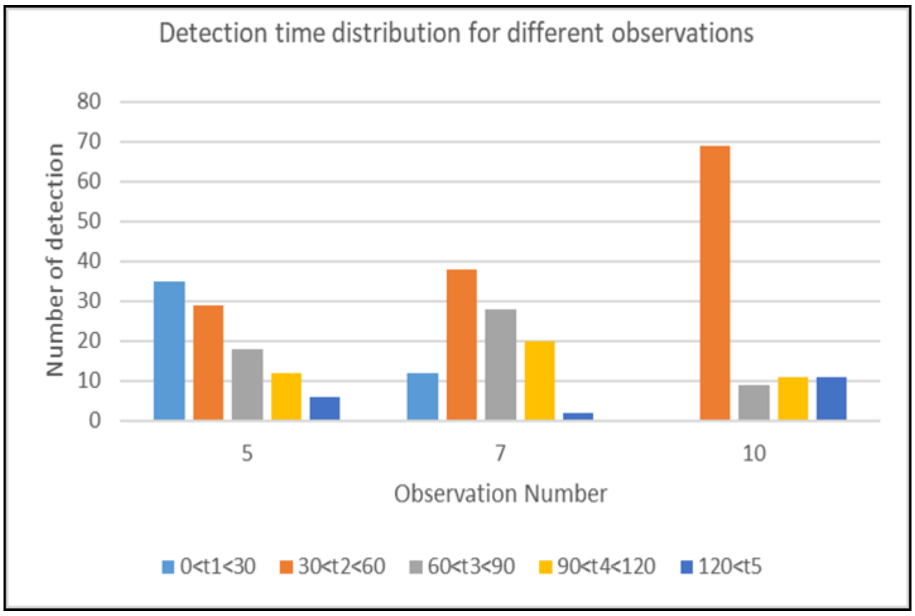


Figure 6.2: Detection time distribution for different observations

7 Conclusion and Future work

This work has presented not only a source-based intelligent approach for detecting LSD-DoS attack in SDN environment but also able to identify attack source precisely. We have also used unsupervised ML algorithm “SOM” which is capable of handling the real-time data. Our proposed detection module has the ability to detect the attack in large scale networks though we have tested on only one OF switch. We have shown how an attacker can easily overflow SDN switches with minimum packet rate and still our module is capable to identify. Experiments in a large scale SDN testbed consisting of commercial hardware switches can demonstrate the feasibility and effectiveness of the detection module. Our work is the novel approach for LSDDoS detection and it can be improved in several ways. In future work, we are going to reduce the detection time and make it much faster than present work. Attacker information needs to be shared with neighbor controllers which can alert other controllers before the attack hits them. We will adapt our detection module for the large networks where many switches and servers will be connected to the controller and for the most common SDN controllers.

Bibliography

[1] Cambiaso, Enrico & Papaleo, Gianluca & Chiola, Giovanni & Aiello, Maurizio "Slow DoS attacks: definition and categorisation", *International Journal of Trust Management in Computing and Communications*, 1. 300-319. 10.1504/IJTMCC.2013.056440.

[2] Lukaseder, Thomas & Ghosh, Shreya & Kargl, Frank. "Mitigation of Flooding and Slow DDoS Attacks in a Software-Defined Network", `https://arxiv.org/abs/1808.05357`, (accessed 23 Jan 2019).

[3] Lee,Dave, "Global internet slows after 'biggest attack in history'", `http://www.bbc.co.uk/news/technology-21954636`, 27 March,2013,(accessed 23 Jan 2019).

[4] Paul Goransson, Chuck Black "Software Defined Networks: A Comprehensive Approach", *Elsevier, 2014*.

[5] "The Open Networking Foundation," OpenFlow Switch Specification Version 1.3.0", `https://www.opennetworking.org`, (accessed 23 Jan 2019).

[6] "Message Layer Definition, OpenFlow Messages", `http://flowgrammable.org/sdn/openflow/message-layer/`, (accessed 23 Jan 2019).

[7] Jia, Su & Jin, Xin & Ghasemiesfeh, Golnaz & Ding, Jiaxin & Gao, Jie. "Competitive analysis for online scheduling in software-defined optical wan.", 2017, 1-9. 10.1109/INFOCOM.2017.8056969.

[8] Jang, RhongHo et al. "RFlow+: An SDN-based WLAN monitoring and management framework.", *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications (2017): 1-9*.

[9] Sonchack, John & J. Aviv, Adam & Keller, Eric & Smith, Jonathan. "Enabling practical software-defined networking security applications with ofx", doi: 10.14722/ndss.2016.23309.

[10] Pascoal, Túlio & Dantas, Yuri & E. Fonseca, Iguatemi & Nigam, Vivek. *"Slow TCAM Exhaustion DDoS Attack."*, 2017. 17-31. 10.1007/978-3-319-58469-0_2.

[11] Cao, Jiahao & Xu, Mingwei & Li, Qi & Sun, Kun & Yang, Yuan & Zheng, Jing. "Disrupting SDN via the Data Plane: A Low-Rate Flow Table Overflow Attack.", 2017. 10.1007/978-3-319-78813-5_18.

[12] Lukaseder, Thomas & Maile, Lisa & Erb, Benjamin & Kargl, Frank. "SDN-Assisted Network-Based Mitigation of Slow DDoS Attacks", `https://arxiv.org/abs/1804.06750`, (accessed 23 Jan 2019).

[13] K. Hong, Y. Kim, H. Choi and J. Park "SDN-Assisted Slow HTTP DDoS Attack Defense Method", *IEEE Communications Letters, vol. 22, no. 4, pp. 688-691, April 2018.* doi:10.1109/LCOMM.2017.2766636.

[14] C. Zhang, Z. Cai, W. Chen, X. Luo, & J. Yin "Flow level detection and filtering of low-rate DDoS", *Computer Networks,vol. 56, no. 15, pp. 3417-3431, 2012* .

[15] Siyi Qiao, Chengchen Hu, Xiaohong Guan, & Jianhua Zou "Taming the Flow Table Overflow in OpenFlow Switch.", *ACM SIGCOMM Conference (SIGCOMM '16). ACM, New York, NY, USA, 591-592.*, `https://doi.org/10.1145/2934872.2959063`.

[16] Hanqing Zhu, Mingwei Xu, Qing Li, Jun Li, Yuan Yang & Suogang Li "MDTC: An efficient approach to TCAM-based multidimensional table compression", *2015 IFIP Networking Conference (IFIP Networking), Toulouse, 2015, pp. 1-9*, doi: 10.1109/IFIPNetworking.2015.7145306.

[17] Ying Qian, Wanqing You & Kai Qian "OpenFlow flow table overflow attacks and countermeasures", *2016 European Conference on Networks and Communications (EuCNC), Athens,2016*, doi: 10.1109/EuCNC.2016.7561033.

[18] S. Sezer et al., *"Are we ready for SDN? Implementation challenges for software-defined networks", IEEE Commun. Mag., vol. 51, no. 7, pp. 36–43, Jul. 2013.*

[19] M. D. Yosr Jarraya and T. Madi, "A survey and a layered taxonomy of software-defined networking,", *IEEE Commun. Surveys Tuts., vol. 16, no. 4, pp. 1955–1980, 4th Quart. 2014.*

[20] Q. Yan, F. R. Yu, Q. Gong and J. Li "Software-Defined Networking (SDN) and Distributed Denial of Service (DDoS) Attacks in Cloud Computing Environments: A Survey, Some Research Issues, and Challenges,", *IEEE Communications Surveys & Tutorials, vol. 18, no. 1, pp. 602-622, First quarter 2016.*, doi: 10.1109/COMST.2015.2487361

[21] Doan Hoang, "Software Defined Networking ? Shaping up for the next disruptive step?", *Australian Journal of Telecommunications and the Digital Economy. 3. 48. 10.18080/ajtde.v3n4.28.*

[22] T. Kohonen, "The self-organizing map,", *Proceedings of the IEEE, vol. 78, no. 9, pp. 1464–1480, 1990*

[23] Box, Joan Fisher. "Guinness, Gosset, Fisher, and Small Samples. Statist. Sci. 2", *(1987), no. 1, 45–52.* doi:10.1214/ss/1177013437.
`https://projecteuclid.org/euclid.ss/1177013437`,

[24] P. Wette, M. Dräxler and A. Schwabe, "Maxinet: Distributed emulation of software-defined networks," in *2014 IFIP Networking Conference,* pp. 1–9, June 2014

[25] *"Description of the onos controller"* `https://www.onosproject.org`

[26] *"Description of the OpenDayLight controller"*, `https://www.opendaylight.org.`

[27] Slowloris, "The LSDDoS Botnet Simulator", (accessed 23 Jan, 2019) `https://github.com/llaera/slowloris.pl`