单位代码 **10006**  

课程名称 **高等计算机网络**

分 类 号 **TP311.1**

****

中期报告

基于SDN的DDoS攻击防御实现

|  |  |
| --- | --- |
| 学院名称 | 计算机学院 |
| 团队成员 | 温雅楠SY2006345 |
| 团队成员 | 武仕沛ZY2006357 |
| 团队成员 | 王宇翔ZY2006160 |

2020年 11 月24日

目录

[1. 研究内容 1](#_Toc57309500)

[2. 已完成工作 1](#_Toc57309501)

[2.1环境部署 1](#_Toc57309502)

[2.2模块设计与实现 8](#_Toc57309503)

[2.2.1合法请求回送报文生成模块 8](#_Toc57309504)

[2.2.2非法攻击流量生成模块 10](#_Toc57309505)

[2.2.3 packet-in协调转发模块 12](#_Toc57309506)

[2.2.4小型服务器检测模块 16](#_Toc57309507)

[2.3实验设计与结果分析 16](#_Toc57309508)

[2.3.1网络连通性及服务器响应测试 16](#_Toc57309509)

[2.3.2链路带宽攻击与防御测试 18](#_Toc57309510)

[2.3.3单点攻击测试 21](#_Toc57309511)

[3. 关键技术与难点 23](#_Toc57309512)

[3.1链路带宽占用攻击模拟 23](#_Toc57309513)

[3.2 packet-in报文协调转发策略 25](#_Toc57309514)

[4. 作业分工与进度安排 27](#_Toc57309515)

[5. 参考资料 28](#_Toc57309516)

## 研究内容

分布式拒绝服务攻击（DDoS）通过大量合法分布式节点对服务器发送请求，从而淹没服务器临近链路带宽或耗尽服务器本身计算资源，最终使正常用户无法获得服务器的响应。

由于现行网络本身具有分布式的特点，因此在面对DDoS攻击时，往往难以及时协调一致的进行响应。通常分布式的网络设备是不具备状态的，也很难进行集中管理以记录状态，因此在收到转发包时，直接采取查表的方式进行转发，而不对数据流进行分析，这就给DDoS攻击流留有可乘之机。因此，当网络检测到DDoS攻击时，被攻击服务器临近链路已经被消耗了大量带宽，且服务器也已经收到了大量非法的请求。可见传统的网络在面临该攻击行为时，其反制措施往往有一定后滞性。

传统的采用分布式协议的网络如不能如期对DDoS攻击进行有效的防御，那么便考虑集中控制式的网络模型。因此，本实验采用SDN网络模型，结合mininet搭建起虚拟网络环境，控制器采用FloodLight，控制器和交换机之间采用OpenFlow协议传输报文和下发流控制规则，网络流量监视器采用sFlow进行监控。

本实验的工作可以分为两大部分，一是通过在mininet中搭建虚拟网络环境，并结合FloodLight开发，通过源码级的调试，来测试诸如控制器模块、报文交换模块、链路发现模块、拓扑管理模块和转发模块，进而对控制器有一个全面而详细的认识，并结合网络流量监测工具sFlow对网络流量进行实时监控，通过设置转发规则并查看前后流量变化，从而对SDN网络架构及其工作流程有一个宏观的认识。

另一部分的工作内容在于DDoS的攻防对抗上，正如前文所述，DDoS攻击可分为两大类，一是对服务器临近链路带宽的占用，二是对服务器本身计算资源的耗费。因此，这部分工作重点就在于如何设计攻击流量以达到占用链路带宽和耗尽服务器计算资源的效果，以及设计相应的算法或者逻辑规则，并部署到控制器上，针对这二者的攻击，起到良好的防御效果。

## 已完成工作

### 2.1环境部署

本实验采用Windows+Linux双系统联调的方式进行。Windows上部署FloodLight，并在IDEA集成开发环境下进行开发。Linux上部署mininet实现虚拟网络组网，部署sFlow服务，实现网络流量监控。下面是具体的环境清单：

|  |  |  |
| --- | --- | --- |
| Windows10 64位  家庭版 1909 | Java环境 | Jdk 1.8.0\_271 |
| 集成开发环境 | IntelliJ IDEA 2020.2.2 |
| 虚拟机 | Vmware Pro 15.5.6 |
| 网络控制器 | Floodlight v0.9 src |
| Ubuntu 18.04  LTS 64位 | Java环境 | Jdk 1.8.0\_272 |
| Python环境 | Python 2.7.17 |
| 组网软件 | mininet 2.2-2 |
| 流量监控 | sFlow 3.0-1529 |

由于采用主机与虚拟机联调开发，因此需要在虚拟机上配置网关，使得两边网络联通。具体做法是在Vmware上配置NAT地址映射，虚拟机的网关设置为192.168.137.2，子网掩码为255.255.255.0，Linux下设置本机ip为192.168.137.181，然后测试双方连通性，双方连通性测试如图2-1和2-2所示。

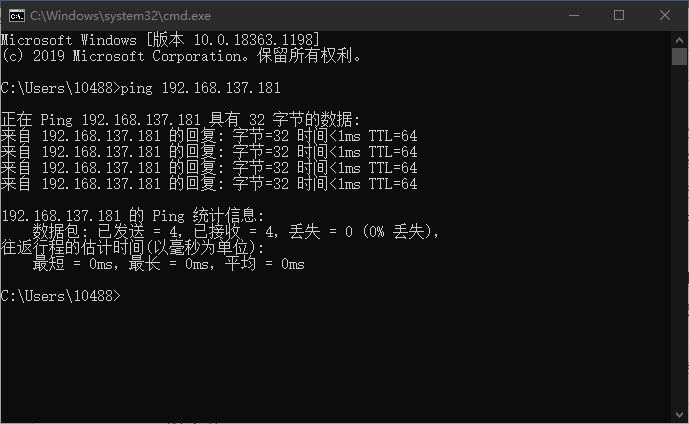


图2-1 主机ping虚拟机

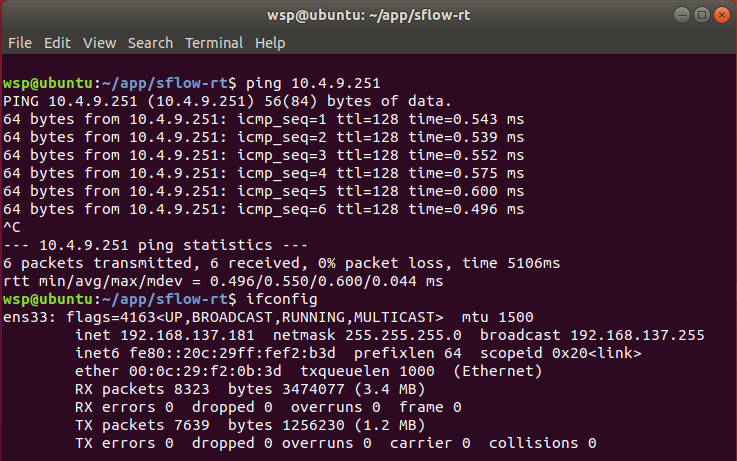


图2-2 虚拟机ping主机

FloodLight控制器开发，本次实验不仅需要会使用FloodLight对网络进行管理，而且还要能够开发相应模块，以针对DDoS攻击实施转发规则的下发。因此本次实验FloodLight采用源码开发，配合maven对项目进行管理、构建。FloodLight源码可通过git clone git://github.com/floodlight/floodlight.git 命令下载，之后在IDEA中导入该工程，即可进行开发。初始版本可直接进行编译运行，FloodLight自带前端模块，可通过可视化的形式对所管理网络进行展示，初始启动环境配置参数在resources/floodlightdefault.properties文件中，这里列举一二：

|  |  |
| --- | --- |
| net.floodlightcontroller.restserver.RestApiServer.port | 8080 |
| net.floodlightcontroller.core.FloodlightProvider.openflowport | 6633 |
| net.floodlightcontroller.core.FloodlightProvider |  |
| ...... |  |

第一行参数为北向访问端口号，除了浏览器界面通过该端口对网络进行查看意外，流表的查看/下发也是通过该端口传输的。第二行参数为南向访问端口号，ovs交换机与控制器进行绑定就是通过该端口实现。第三行之后是控制器默认支持的模块，包括常见的链路发现模块、拓扑管理模块、转发模块等。FloodLigtht开发界面和启动界面分别如图2-3、2-4所示。

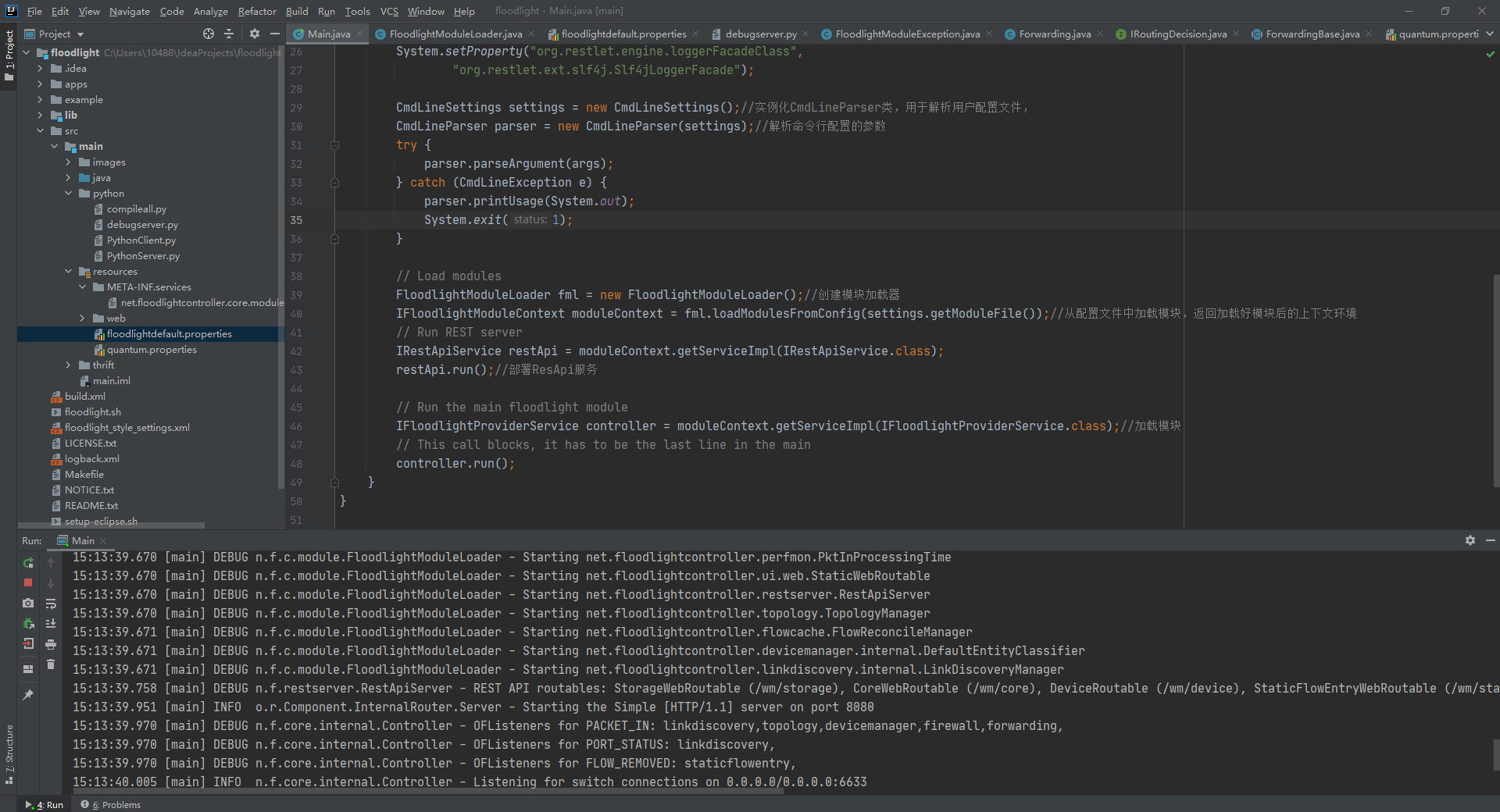


图2-3 FloodLight开发界面

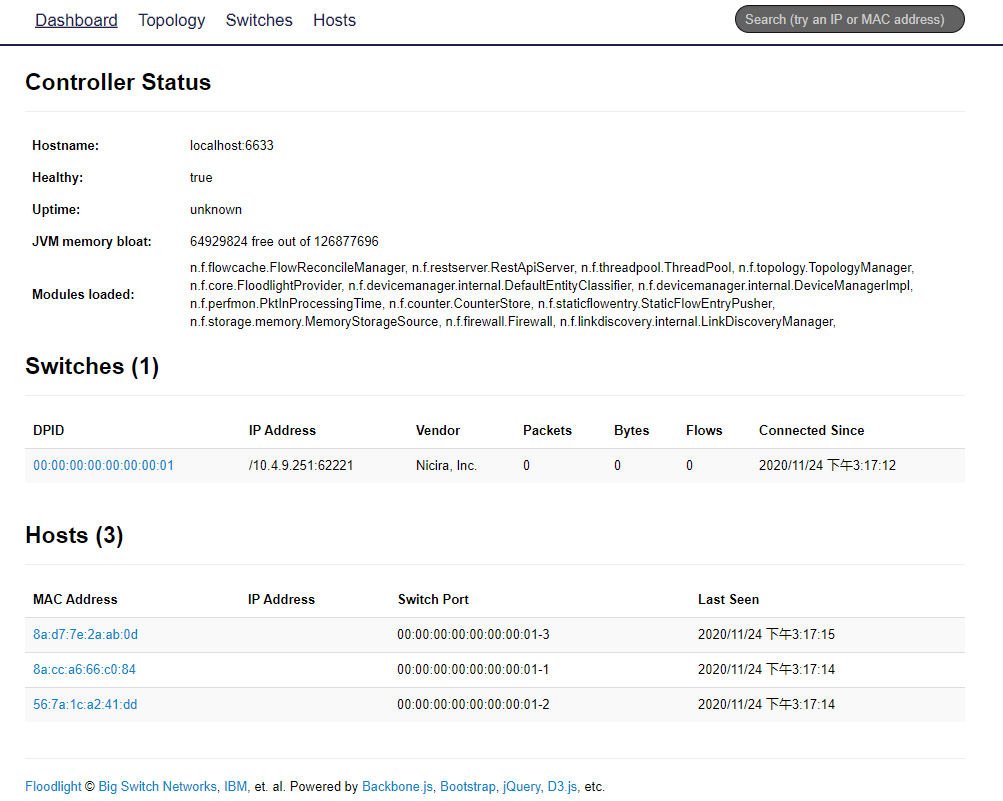


图2-4 FloodLight前端可视化界面

Linux环境下，通过apt-get方式安装mininet组网软件，命令为sudo apt-get install mn，Java和Python环境以及sFlow监控软件通过源码包方式安装。

mininet组网软件使用方式比较简单，可通过命令式/交互式的方式生成期望的拓扑结构，这里以一个中心节点出度为8，深度为2的树形网络拓扑结构创建为例，使用的命令为：sudo mn --switch ovsk --topo tree,depth=2,fanout=8 --controller=remote,ip=10.4.9.251,port=6633，各参数的意义如下：

sudo：由于虚拟组网环境需要调用到内核网卡接口，因此需要管理员权限运行。

--switch：指定网络中交换机的类型，由于该实验交换机需要与控制器进行交互，因此需选用能够支持OpenFlow协议的交换机，ovsk中的ovs即代表Open vSwitch。

--topo：网络拓扑结构类型，常见的有single（单点）、linear（总线）、tree（树形）结构，根据选择的类型不同，需要指定具体的细分参数，如树形结构中需指明树深度，节点度数。

--controller：指定控制器来源，需指明ip地址和端口号。

创建过程如图2-5所示，另外，创建的网络拓扑结构可视化如图2-6所示。

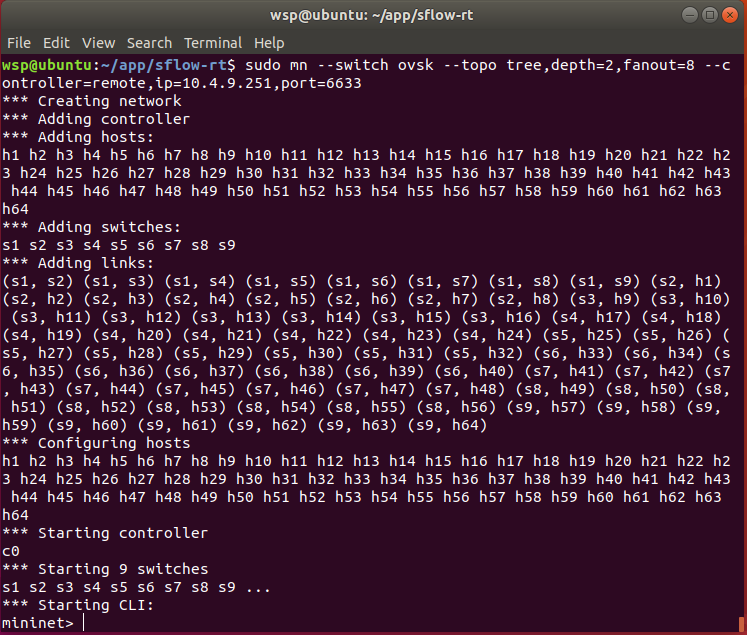


图2-5 mininet搭建树形组网图

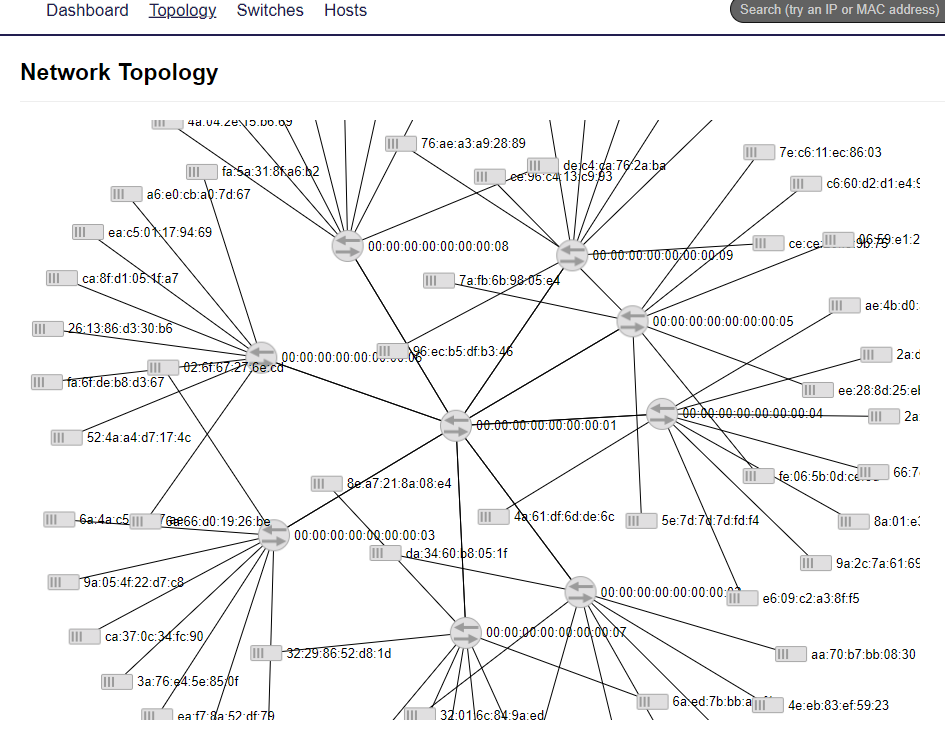


图2-6 FloodLight拓扑结构界面

mininet除了可以进行虚拟网络搭建以外，还可以针对交换机、主机进行单独管理，例如ping报文测试，这也为后面的攻击行为提供了环境基础。使用方式实在mininet交互式命令行中，通过xterm -name单独对打开的节点进行管理，（-name代表交换机或主机节点），这里以xterm h1为例，打开h1主机的管理窗口，并进行ping报文的测试，被ping主机为h32（10.0.0.32），ping命令结果如图2-7所示。

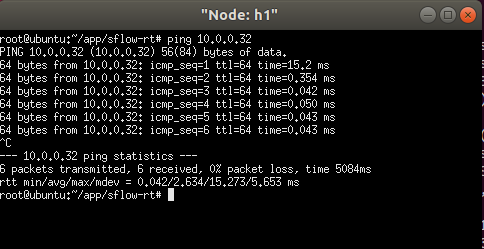


图2-7 虚拟主机节点通过xterm进行ping通测试

流量监控软件sFlow可以实现对网络中单个或多个被管交换机进行流量监控，这取决于配置的sFlow-agent的数量。首先启动sFlow服务端，命令为：./sflow-rt/start.sh。之后需要针对被管交换机设置sFlow-agent，设置了agent的交换机会将自身链路接口处的流量信息上传给sFlow-agent，agent配置命令为：sudo ovs-vsctl -- --id=@sflow create sflow agent=eth0 target=\"127.0.0.1:6343\" sampling=10 polling=20 -- -- set bridge s1 sflow=@sflow。命令参数意义如下：

agent：指定采样的网卡，由于虚拟组网环境中，所有的流量都会经过虚拟机的主网卡，因此网卡为eth0。

target：sFlow服务端地址，默认端口号为6343。

sampling：采样率，即每隔N个Packet采样一次。

polling：轮询时间，即每隔N秒轮询一次。

bridge：被监控的网络设备，这里以h1所在的交换机s1为例，待会儿以h1主机进行ping报文测试，查看流经s1的流量。需说明的是，如果要针对全网环境测试的话，该命令需要多次配置，以覆盖所有交换机/路由器。

服务端，客户端都配置完成以后，便可打开服务端界面，默认端口号为8008，总览界面如图2-8所示。

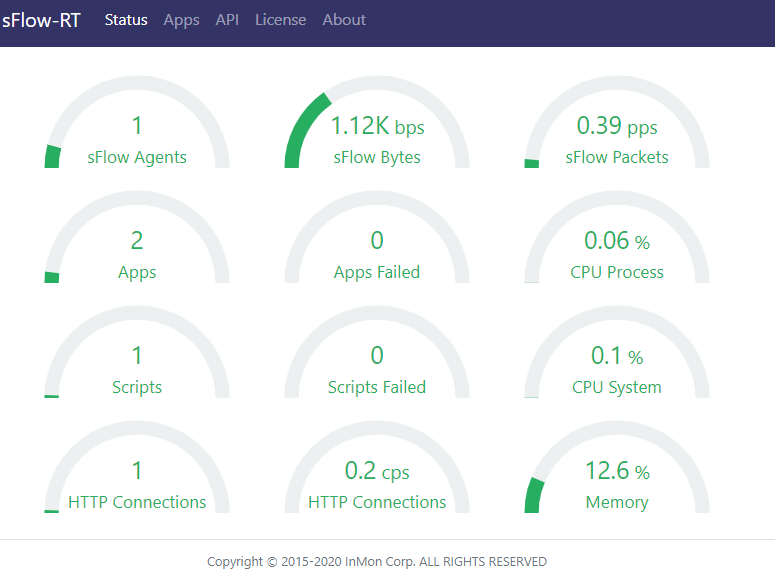


图2-8 sFlow流量监控仪表盘

这时在mininet中执行h1 ping h32，查看sFlow流量情况，通过设置规则为“ipsource，ipdestination”查看该指标下的流量情况，流量结果如图2-9所示，可见蓝色代表h1主机ping h32主机所产生报文的流量，红色代表h32主机返回给h1主机的报文流量。

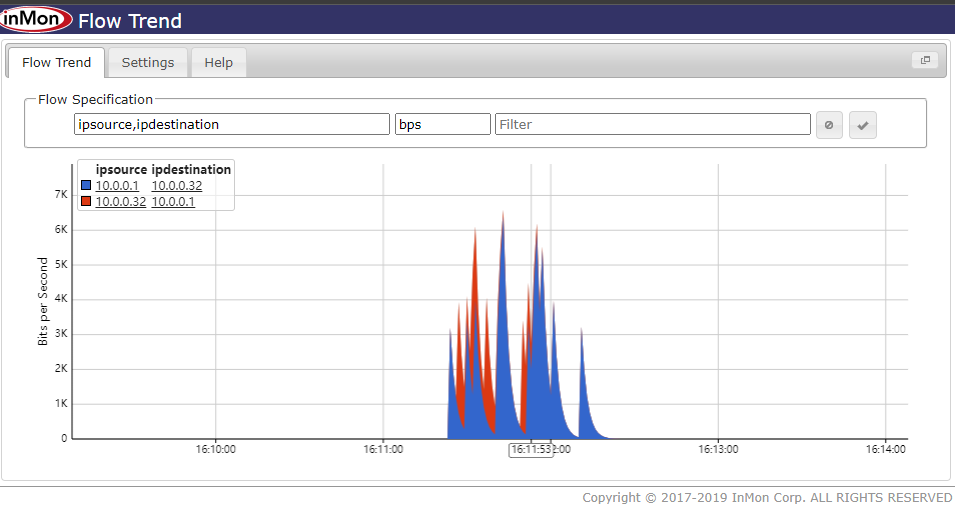


图2-9 sFlow监测ping报文所占流量带宽

### 2.2模块设计与实现

#### 2.2.1合法请求回送报文生成模块

合法请求回送报文生成模块用于检验网络的连通性，确切的说是目标服务器的响应报文是否能够正常返回客户端，在配置了DDoS防御机制以后。通过对合法请求回送报文的监控，判断该防御机制是否影响到了正常的请求，同时还可以随非法请求流量生成模块一起启动，通过检测非法攻击流量对合法请求回送报文的影响程度，来判断DDoS攻击是否有效。

合法请求回送报文生成模块通过模拟请求回送报文的产生，另目标ip地址在10.0.0.1-10.0.0.64范围内随机产生，并设置请求回送报文的发送周期、重传次数、等待时间，以最大限度的模拟真实且合理的网络请求。

合法请求回送报文生成模块的代码，由Python编写。

|  |
| --- |
| #coding=utf-8  import sys  import getopt  import logging  logging.getLogger("scapy.runtime").setLevel(logging.ERROR)  from scapy.all import IP, TCP, sr  from random import randrange  #目标ip地址产生函数，生成范围为10.0.0.start-10.0.0.end  def generateDestinationIP(start, end):  first = 10  second = 0  third = 0  ip = ".".join([str(first), str(second), str(third), str(randrange(start,end))])  return ip  def main(argv):  #从启动命令中获取inter，retry，timeout，fixed参数，用于在向目标服务器发送请求回送报文设置发送间隔、超时重传和等待时间，以及目标ip地址是否固定为10.0.0.64  try:  opts, args = getopt.getopt(sys.argv[1:], 'i:r:t:f:', ['inter=','retry=','timeout=','fixed='])  except getopt.GetoptError:  sys.exit(-1)  for opt, arg in opts:  if opt =='-i':  inter = float(arg)  elif opt =='-r':  retry = int(arg)  elif opt =='-t':  timeout = int(arg)  elif opt =='-f':  fixed = bool(int(arg))  if inter == '' or retry == '' or timeout == '' or fixed == '':  sys.exit(-1)  num=100  #循环发送次数  for i in xrange(num):  #如果设值fixed=1（固定），则目标ip地址设值为10.0.0.64，否则在10.0.0.2——10.0.0.64范围内随机产生  if(fixed == True):  desIp = "10.0.0.64"  else:  desIp = generateDestinationIP(2, 65)  print("get reply from : ip[+"+ desIp +"]")  a,b = sr(IP(dst = desIp)/TCP(),inter = inter, retry = retry, timeout = timeout)  #生成请求响应报文，数据包发送间隔inter，无应答时重发次数retry，数据包等待时间timeout  if \_\_name\_\_ == '\_\_main\_\_':  main(sys.argv) |

注意，在针对服务器周边链路带宽占用和服务器计算资源消耗这两种不同的DDoS攻击场景时，目标主机的ip地址、等待时间设值有所不同，具体的，由于前者的攻击类型会导致网络中部分或全部的链路带宽被耗尽，这时候正常请求报文在全网范围内都会受到不同程度的影响，因此设置目标ip地址为10.0.0.2-10.0.0.64就是为了检验h1主机（10.0.0.1）对网络中其他服务器节点的请求回送情况，等待时间视不同链路带宽占用情况灵活设值。而在针对服务器计算资源耗费型的DDoS攻击场景下，这时候请求回送报文的瓶颈就在于目标主机及其直连的交换机上了，此时目标ip地址设值为固定，以h64（10.0.0.64）为例，当启动攻击模块时，由于h64主机接收到大量非法请求报文，Cpu忙于计算，势必会导致合法请求报文处理不及时，因此，除了目标ip地址固定外，还应设值不同的等待时间，以评估目标服务器的处理能力，和针对此种攻击场景下的防御措施的有效性。

基于此，合法请求回送报文生成模块启动命令参数设置inter（发送间隔）、retry（重传次数）、timeout（等待时间）、fixed（目标ip是否固定），通过以上参数，可实现针对不同攻击场景下的所应当被测量的合法请求报文生成模块。

#### 2.2.2非法攻击流量生成模块

非法攻击流量生成模块用于实施对网络链路带宽的攻击和对主机计算资源的攻击。针对网络链路带宽的攻击，设值源和目标ip地址在10.0.0.1-10.0.0.64范围内随机生成，以向controller发送大量的packet-in报文（注意，这里呈现的是占用链路带宽的另一种形式，与常规形式有所不同，这部分在3.1节中详细展开）。而对服务器计算资源的攻击，则设值目标ip地址固定，以实现大量请求报文打在同一个主机节点上。这两种攻击同样可通过设置不同的启动命令参数-f（fixed）实现。

非法攻击流量生成模块的代码如下，由Python编写。

|  |
| --- |
| #coding=utf-8  import sys  import time  import getopt  from os import popen  import logging  logging.getLogger("scapy.runtime").setLevel(logging.ERROR)  from scapy.all import sendp, IP, UDP, Ether  from random import randrange  #生成范围为10.0.0.start——10.0.0.end范围内的ip地址  def generateSourceIP(start, end):  first = 10  second = 0  third = 0  ip = ".".join([str(first), str(second), str(third), str(randrange(start,end))])  return ip  def generateDestinationIP(start, end):  first = 10  second = 0  third = 0  ip = ".".join([str(first), str(second), str(third), str(randrange(start,end))])  return ip  def main(argv):  #获取启动参数，start，end用于控制生成ip地址的范围，fixed设置目标ip地址是否固定  try:  opts, args = getopt.getopt(sys.argv[1:], 's:e:f:', ['start=','end=','fixed='])  except getopt.GetoptError:  sys.exit(-1)  for opt, arg in opts:  if opt =='-s':  start = int(arg)  elif opt =='-e':  end = int(arg)  elif opt =='-f':  fixed = bool(int(arg))  if start == '' or end == '' or fixed == '':  sys.exit(-1)  #总共5轮攻击，每轮Dos攻击500次  for i in range (1, 5):  interface = popen('ifconfig | awk \'/eth0/ {print $1}\'').read()  #获取本机网卡接口，sendp发包通过该接口实现  for i in xrange(0, 500):  if(fixed == True):  desIp = "10.0.0.64"  else:  desIp = generateDestinationIP(start, end)  packets = Ether() / IP(dst = desIp, src = generateSourceIP(start,end)) / UDP(dport = 1, sport = 80)  #如果是链路带宽攻击，则设值目标ip地址随机，如果是对主机计算资源攻击，则设值目标ip地址固定  print(repr(packets))  sendp(packets, iface = interface.rstrip(), inter = 0.025)  #发送间隔为0.025秒，用于模拟请求泛洪  time.sleep (10)  if \_\_name\_\_=="\_\_main\_\_":  main(sys.argv) |

注意，与合法请求回送报文生成模块的不同之处在于，这里使用的不再是sr()函数，而是sendp()函数，前者代表了一个请求加接收响应的完整过程，后者仅实现发送过程，不关心响应报文。并且发送函数的发送间隔（inter）设值的更短，通常为0.025秒，以模拟请求泛洪，给链路带宽和服务器计算资源造成巨大压力。

#### packet-in协调转发模块

packet-in报文是用于当网络设备不知道当前的数据包该如何转发时向控制器发送的路径请求计算报文，即交换机或路由器没有相关路由转发表项时就会向控制器请求生成一个匹配当前数据报目的地址的转发表项。然而，如果交换机（例如本实验中S2）接收到了大量ip数据报，且此刻这些目的ip地址都还未出现在路由转发表项中，就会向控制器发出大量的packet-in报文，造成该交换机与控制器的链路发生拥塞，抑或是造成控制器本身处理负载过大，此时，如果合法的请求报文经过此交换机，恰巧该交换机也没有匹配该报文目的地址的转发表项时，就会因为其与控制器的链路发生拥塞从而导致该packet-in报文迟迟无法得到控制器的处理，相应的转发表项也就得不到计算，这样一旦该请求报文的rtt超时以后，就会重传报文，更严重情况是该请求永远也得不到应答。

既然单个交换机与控制器的链路负载很大，则可以考虑采用将负载分摊到不同交换机上以解决该问题。为了评估何时将自身的负载转移到其他网络设备上，需要有一个指标来衡量此时交换机的负载率，如果负载率大一某一阈值a时，将会导致该交换机的北向接口拥塞，便采取措施将这些数据报无条件转发到其他交换机上，由其他交换机向控制器发送packet-in报文请求路径计算，当负载率下降到某一阈值b时（b一般小于a），则停止无条件转发行为，由自身请求控制器处理该报文。

为了实现上述方案，需要对FloodLight中的ForwardingBase.java（该模块是所有与交换机数据转发相关的基类）进行改造，其中原本的receive函数用于处理与交换机交互时所接收到的原始的PACKET\_IN报文，其处理逻辑如下所示：

|  |
| --- |
| @Override  public Command receive(IOFSwitch sw, OFMessage msg,  FloodlightContext cntx) {  switch (msg.getType()) {  case PACKET\_IN:  IRoutingDecision decision = null;  if (cntx != null) {  decision = IRoutingDecision.rtStore.get(cntx, IRoutingDecision.CONTEXT\_DECISION);  }  return this.processPacketInMessage(sw,  (OFPacketIn) msg,  decision,  cntx);  default:  break;  }  return Command.CONTINUE;  } |

注意这里如果对于接收到的是PACKET\_IN报文类型，控制器所进行的操作，可以看到其最终调用了processPacketInMessage函数，该方法是抽象方法，实现它的类是Forwarding.java，实现逻辑是根据从控制器的路由规则上下文环境中（IRoutingDecision.CONTEXT\_DECISION）计算该数据报的转发表项，并返回相应流表给交换机。现为衡量当前交换机的packet-in处理负载率，设置初始复杂率packetInDutyFactor（类型为int）为0。当接收到packet-in报文时，计算前后接收到的packet-in报文的时间差，当时间差小于packet-in处理时间的一半时，复杂率加1，否则维持不变，同时在另一个线程thread里，每当经过一个packet-in处理时间时，复杂率减一。在接受packet-in报文时，比较当前负载率是否大于阈值a，如果大于，则下发无条件转发流表，该流表作用是交换机将收到的任何数据报直接向出端口转发，而不是向控制器发送packet-in报文。同理，在thread线程中，当负载率降低到b时，收回该流表，即取消该交换机的无条件转发行为。

下面是对原始ForwardingBase.java改进的代码说明，首先是针对reveive函数做如下变动：

|  |
| --- |
| @Override  public Command receive(IOFSwitch sw, OFMessage msg,  FloodlightContext cntx) {  switch (msg.getType()) {  case PACKET\_IN://处理packet-in报文情况  if (packetInDutyFactor == 0) {//如果当前负载率为0，则更新为1，  lastPacketInTime = System.currentTimeMillis();  packetInDutyFactor = 1;  } else if (DutyFactorAddTimeDis() < 25) {//如果两次接收packet-in报文时间差小于该报文处理时间的一半，则负载率+1  packetInDutyFactor++;  }    if (packetInDutyFactor == BUSYTHRESHOLD) {//当负载率达到高负载阈值时，执行curl命令，调用控制器北向接口的rest api，注入一条无条件转发流表  Curl.execCurl(new String[]{"curl", "-X", "POST", "-d", "@ddos\_defend.json",  "http://127.0.0.1:8080/wm/staticflowentrypusher/json"});  } else {//否则，认为负载率较低，仍按之前方式计算转发路径  IRoutingDecision decision = null;  if (cntx != null) {  decision = IRoutingDecision.rtStore.get(cntx, IRoutingDecision.CONTEXT\_DECISION);  }  return this.processPacketInMessage(sw,  (OFPacketIn) msg,  decision,  cntx);  }  default:  break;  }  return Command.CONTINUE;  } |

新增线程，用于监测负载率是否降低到轻负载阈值。

|  |
| --- |
| new Thread(() -> {  while (true) {  try {  Thread.sleep(50);  } catch (InterruptedException e) {  e.printStackTrace();  }  packetInDutyFactor = packetInDutyFactor > 0 ? packetInDutyFactor- 1 : 0;//每隔packet-in报文处理时间，负载率减1  if (packetInDutyFactor == IDLETHRESHOLD) {//如果降低到轻负载率，则删除之前下发的流表  Curl.execCurl(new String[]{"curl", "-X", "DELETE", "-d", "\"{\\\"name\\\":\\\"no\_forward\_process\\\"}\"",  "http://127.0.0.1:8080/wm/staticflowentrypusher/json"});  }  }  }); |

Curl工具类，调用windows的cmd，向控制器发送curl请求，即控制器提供的rest api，用于增删流表。

|  |
| --- |
| public class Curl {  //通过cmd命令向controller发送curl命令，用于下发流表  public static String execCurl(String[] cmds) {  ProcessBuilder process = new ProcessBuilder(cmds);  Process p;  try {  p = process.start();  BufferedReader reader = new BufferedReader(new InputStreamReader(p.getInputStream()));  StringBuilder builder = new StringBuilder();  String line = null;  while ((line = reader.readLine()) != null) {  builder.append(line);  builder.append(System.getProperty("line.separator"));  }  return builder.toString();  } catch (IOException e) {  System.out.print("error");  e.printStackTrace();  }  return null;  }  } |

无条件转发规则的设置，如下，当交换机s2收到该转发规则时，生成一条流表项，从2号端口进入的数据报，都直接向1号端口转发，设置优先级100，这样即使收到未匹配流表项的目的ip数据报时，也不会向控制器发送packet-in报文。

|  |
| --- |
| {  "switch":"00:00:00:00:00:00:00:02",  "name":"no\_forward\_process",  "cookie":"0",  "in\_port":"2",  "priority":"100",  "active":"true",  "actions":"output=forward",  "dst\_port":"1"  } |

#### 2.2.4小型服务器检测模块

该模块设计较为简单，通过python命令即可生成一个Http请求响应服务器。设计该模块的意义在于当DDoS攻击发生时（针对链路带宽的攻击和服务器计算资源的攻击），通过发送http请求来检验攻击的影响程度，在部署防御模块以后，再次查看http请求响应情况，来验证防御模块的有效性。

该模块与2.2.1节中设计的合法请求回送报文生成模块其实都可以验证以上两点，不过2.2.1节更注重分析请求响应报文的转发过程，从比较低的层次来验证DDoS攻防模型，而该节中通过一个http服务器，从应用层面上更为直观的验证DDoS攻防模型，可以说二者互为补充。

小型服务器的部署命令及请求命令如下：

|  |
| --- |
| h64: python -m SimpleHttpServer 80  h1: curl http://10.0.0.64 //这里以h64为例 |

### 2.3实验设计与结果分析

#### 2.3.1网络连通性及服务器响应测试

首先启动FloodLight控制程序，java -jar ./target/floodlight.jar，启动过程如图2-10所示。

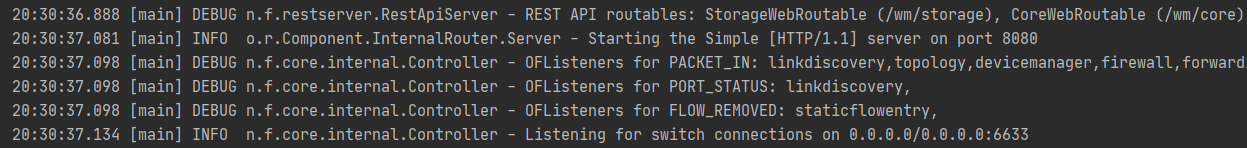


图2-10 FloodLight启动过程

启动组网程序，sudo mn --switch ovsk --topo tree,depth=2,fanout=8 --controller=remote,ip=10.4.9.251,port=6633，上述命令为创建一个树形拓扑网络，深度为2层，树节点度为8，交换器为支持OpenFlow协议的ovs型交换机，控制器连接远端服务器。通过nodes命令查看该网络下设备情况如图2-11所示。

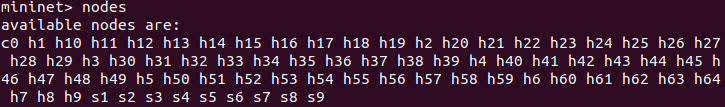


图2-11 nodes查看虚拟网络节点情况

pingall命令进行网络连通性测试，测试结果如图2-12所示。

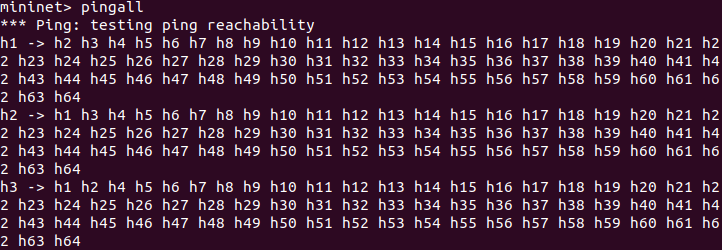


图2-12 pingall网络连通性测试

启动流量检测模块sFlow，结合ping命令，监测到的流量情况如图2-13所示。

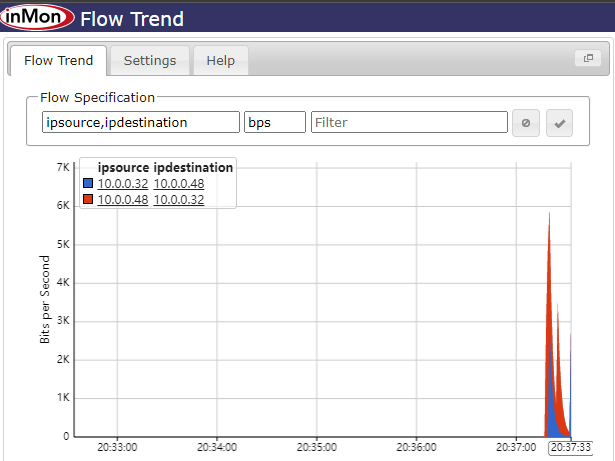


图2-13 流量监控测试

综上，实验所需环境工作正常，可以进行后续DDoS攻防模型测试。

#### 2.3.2链路带宽攻击与防御测试

首先在h1上运行合法请求回送程序，命令为python echo-reply.py -i 0.5 -r 3 -t 0.5 -f 0，该命令后跟参数-i 0.5代表发送间隔0.5s，-r 3代表重传报文次数为3，-t 0.5为等待响应报文时间为0.5s，-f 0代表目标主机ip地址不固定（在10.0.0.2-10.0.0.64范围内随机），运行结果如图2-14所示。可见，正常情况下，请求发送出去后能够及时得到响应，也没有丢包情况。

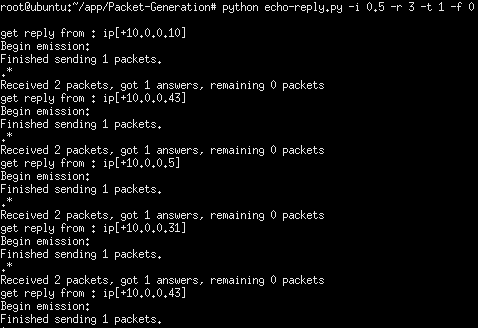


图2-14 正常情况下请求回送报文测试

在sFlow上查看S2（h1位于S2下）的流量情况如图2-15所示。图中前面部分对应发送间隔为1秒的流量状况，后面峰值较高的部分对应的发送间隔为0.5s，无论那种情况，此时网络都能正常的执行数据报的转发。

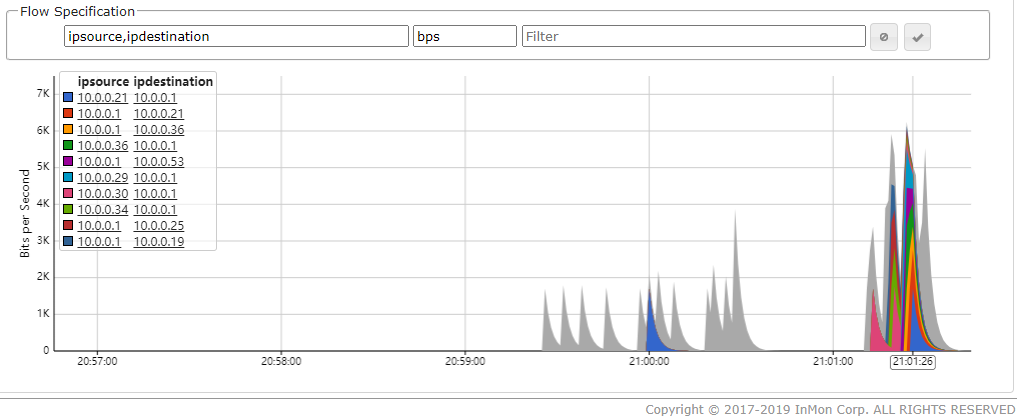


图2-15 请求回送流量监测

此时在h2（h2与h1位于同一交换机下）启动链路带宽攻击程序，命令为python attack.py -s 2 -e 65 -f 0，后跟参数-s和-e一起配合指定源ip地址随机范围（本例中ip范围为10.0.0.1-10.0.0.64），-f 0代表设置目标ip地址不固定，范围同上，程序设置一共攻击5轮，每轮发送请求500下，发送间隔为0.025秒。运行如图2-16所示（真实环境下产生了海量发送日志，图中只截取部分）。



图2-16 链路带宽攻击日志

当链路带宽攻击发生后，再次回到h1上，查看请求回送日志（重传次数3，等待时间0.5s），结果如图2-17所示，可以明显看到发生超时、丢包的情况严重，有的报文达到了最大重传次数也没有得到响应。

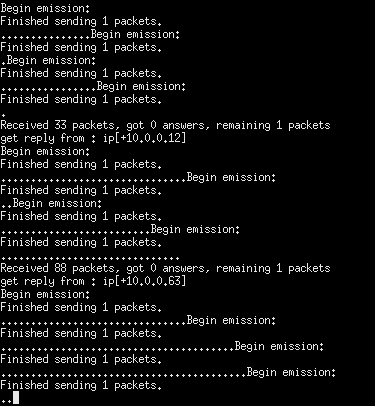


图2-17 链路带宽攻击状态下请求回送报文日志（等待时间0.5s）

当提升报文等待时间为1s时，情况有所好转，结果如图2-18所示，丢包次数减少，但延迟增加。

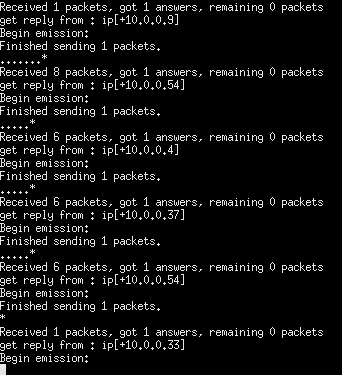


图2-18 链路带宽攻击状态下请求回送报文日志（等待时间1s）

此时再次查看S2端口转发流量，如图2-19所示，发现流量峰值相较之前提升明显，且发送频率极高，另外，由于大量的攻击报文占用了链路带宽，因此很难在流量监控程序中观测到以10.0.0.1为源地址/目的地址的流量，因此导致从h1发送的合法的请求回送流量被淹没。

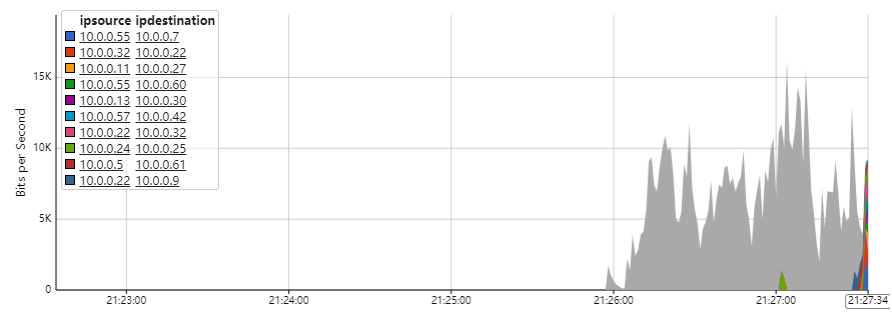


图2-19 链路带宽攻击状态下流量监测视图

此时应用之前对ForwardingBase.java的修改（即开启packet-in协调转发模块），h1的请求回送流量趋于正常，但仍有一定延迟。另外，查看S2交换机上的流表，发现用于协调packet-in转发的流表项已经注入，查看命令为

curl -X GET http://10.4.9.251:8080/wm/staticflowentrypusher/list/00:00:00:00:00:00:00:02/json，命令行结果查看和FloodLight前端显示结果分别如图2-20和2-21所示。

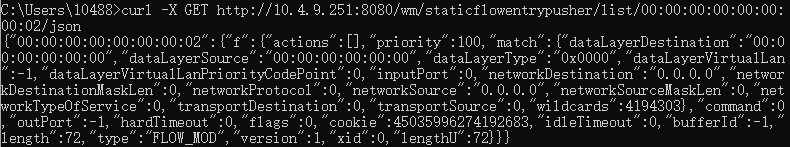


图2-20 curl命令查看s2流表

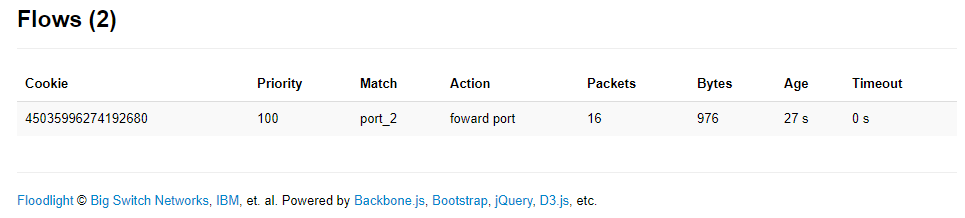


图2-21 FloodLight前端查看s2流表

#### 2.3.3单点攻击测试

在h2上启动attack.py程序，命令为python attack.py -s 2 -e 65 -f 1，-f 1代表请求地址固定为10.0.0.64，源地址仍然随机选择，发送间隔选为0.001s，开启攻击程序后，在h64上使用top命令查看当前进程占用情况，结果如图2-22所示，可见此时h64上为了响应攻击流量，其进程占用率已达到100%。

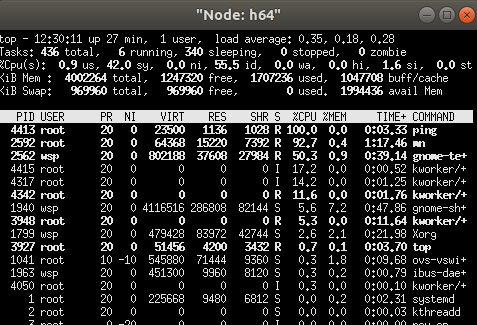


图2-22 h64进程占用情况查看

查看sFlow的流量监控程序，结果如图2-23所示，当攻击流量起来以后，h64所在网络节点s9的接收报文数量稳定在8k/s。

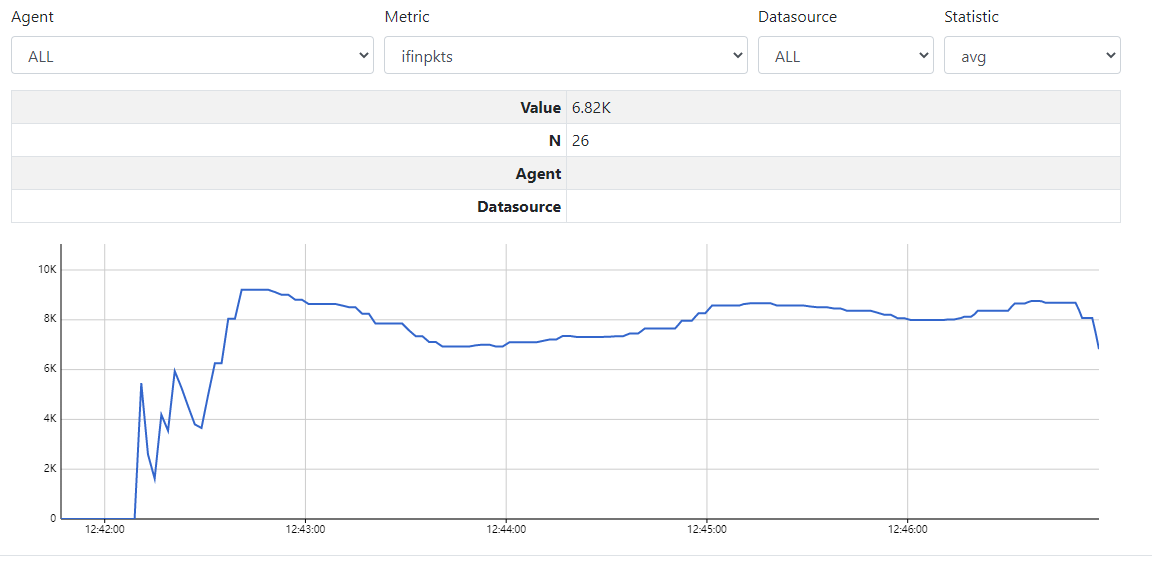


图2-23 h64接收报文速率

运行请求回送程序，查看请求回送报文收发日志，不过结果并不明显，除了响应时间稍稍增加以外，其他数据都很正常，故无法通过该简单的请求回送程序来验证DDoS攻击。为了增加h64节点的处理负载，使用python -m SimpleHttpServer 80启动一个小型Http程序，在h1上通过curl命令请求服务，请求结果如图2-24所示，响应延迟较大。

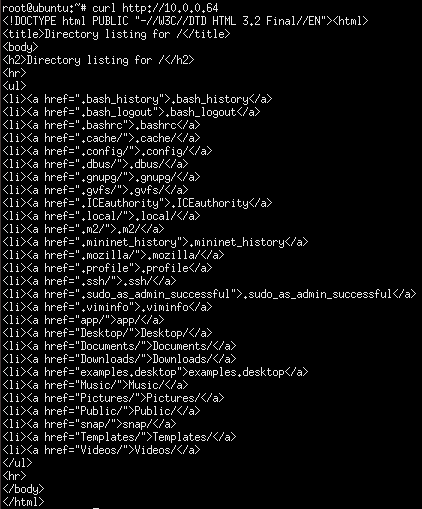


图2-24 h1请求h64服务器

## 关键技术与难点

### 3.1链路带宽占用攻击模拟

这里以图3-1所示的SDN网络架构为例，阐述本节中所要讨论的如何模拟链路带宽占用的攻击场景。

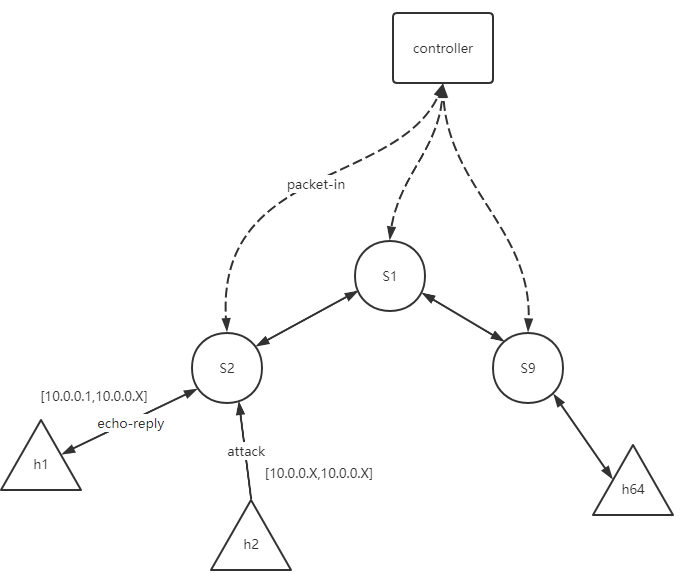


图3-1 模拟链路带宽占用示意图

首先以h1发送请求回送报文为例，源ip地址为自身10.0.0.1，目的ip地址为10.0.0.X（X为2-64任意数），当数据报发送到S2时，由于交换机刚启动，路由表项为空，因此对于这个不知道如何该数据报的交换机来说，只能向控制器发送packet-in报文，请求控制器为其计算一条路径，并返回相应的转发表项。此时packet-in处理过程较为轻松，当h1以正常频率发送数据报时，请求皆能得到回应。

假若此时S2与controller之间产生了大量的packet-in报文传输，那么S2的工作过程就会受到影响，具体的，如果h1发送一个请求报文，且需要S2发送packet-in报文进行处理时，那么大量的非法的攻击流量所导致的packet-in报文就会阻塞该正常的packet-in报文的处理，就会产生延迟，甚至丢包。

其实从这已经可以看出，本实验想要模拟的链路带宽占用并不是通常意义上的链路带宽占用，因为在有限的实验环境下，想要产生使本地网卡接口处带宽耗尽的流量实属不易，因此，本实验换了个角度，针对SDN这种网络架构，不在真实链路上做文章，而是在控制器南向接口与各个网络设备之间的“链路”上进行攻击，事实上，业界已经有论文指出利用packet-in报文来攻击SDN网络设备，进而造成链路带宽占用的假象，本次实验相当于是对该篇论文的复现，并结合所想的处理规则针对这种攻击制定相应解决措施。

基于此，在S2到controller上充斥大量非法的packet-in即可达到效果，因此考虑采用在h2主机上发送大量的源、目的ip地址随机的报文，使得S2产生大量packet-in。

不过由于网络拓扑结构仍相对简单，导致ip地址范围相对有限，因此packet-in的处理速率相对较高，导致较难以消耗掉该条“链路”的带宽。此外，S2接收到controller的回传信息时，会缓存对应转发表项，导致该攻击到后期会失效。针对上述两种问题，通过在ForwardingBase模块中设置处理逻辑，如当收到packet-in报文时，在processPacketInMessage处理方法中执行线程休眠，这里时间设置有相应约束，不能随意增大或减小，需综合数据报所能承受的往返时延，以及控制器负载率来设置。例如h1向h64发送请求回送报文，所经历路径为h1-S2-S1-S9-h64，如果要求报文所能承受的最大往返时延是T ms，则数据报在链路上的传播时延和在交换机上的处理时延总和不能超过T ms，而在本网络环境中，链路的传播时延可忽略不计，因此总的时延来自于packet-in的处理过程。另外，为了实现当请求流量较少时，controller负载较低，而当发生链路带宽攻击时，负载率较高，考虑到请求回送报文的发送间隔时间为500ms，等待时间为500ms，攻击报文的发送间隔为25ms，通过依次选取packet-in的处理时间为200ms，100ms，50ms，10ms，发现，如果处理时间过大，则会超过请求回送报文的往返时延，如果选取过小，则攻击报文也难以阻塞住packet-in报文的处理队列，因此最终选取处理时间为50ms，可以得到较为满意的实验结果。

注意应当关闭交换机对路由表项的缓存功能，这样，在ip地址范围不大的情况下（本例中10.0.0.1-10.0.0.64），仍能长时间让交换机发送packet-in报文。

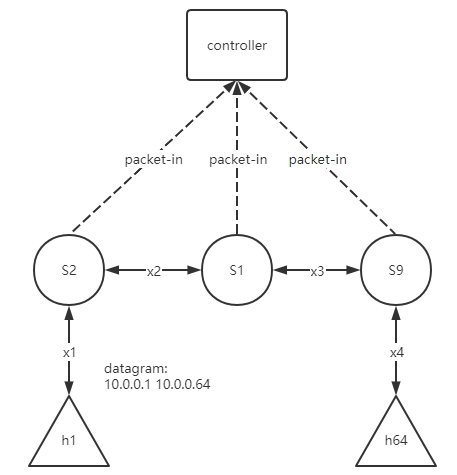


图3-2 报文传输路径及packet-in

### packet-in报文协调转发策略

如果单个交换机由于向控制器发送了大量的packet-in报文而导致本机无法及时处理新的数据报，就会导致本机处理阻塞，因此需要将其收到的新的数据报进行分流。那么会遇到两个问题，一是判定该交换机与其控制器的连接何时处于高负载状态，即判断其忙碌的依据是什么，二是进行数据报无条件转发的策略是什么，向哪一个端口转发，以及什么时候停止无条件转发行为。

为了解决上述问题，首先需要定义一个用于判断交换机是否处于忙碌状态的变量，源码中将其定义为packetInDutyFactor，即交换机负载因子，初始时刻该值为0，该值的变化取决于接收到的packet-in报文和处理该报文的速率。然后需要定义交换机两个状态（忙碌/空闲）的切换阈值，分别为BUSYTHRESHOLD/IDLETHRESHOLD，最后由于需要计算packet-in接收的速率，因此定义两个变量分别是上次接受报文的毫秒数lastPacketInTime和当前接收报文的毫秒数curPacketInTime。

下面以图3-3为例，描述负载因子的变化过程以及状态之间的切换过程。

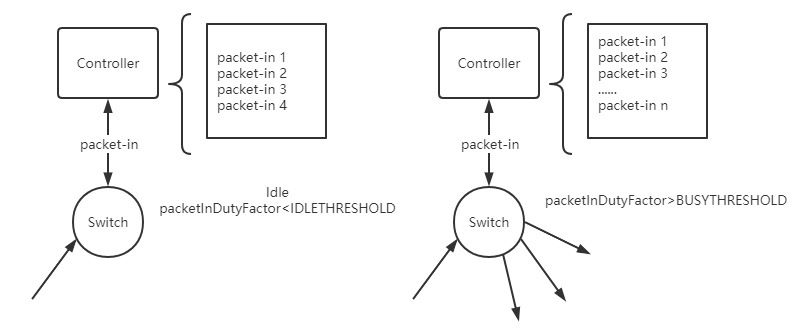


图3-3 packet-in处理导致交换机忙碌/空闲状态的转换示意

首先是packetInDutyFactor负载因子的计算过程，大致逻辑如下所示，当收到第一个packet-in报文时，负载因子为1（此前为0），后续过程中，当前packet-in报文与上一个packet-in报文的接受时间如果小于某一阈值t时，则负载率加1，这里需要说明一下，交换机的负载程度主要通过处理队列的packet-in报文的个数来刻画，因此当packet-in报文数量多时，负载率显然应当增大，那么结合接受速率和处理速率，可以较为合理的定义该t值。在3.1节中，针对链路带宽攻击模拟中，选定了处理速率为50ms，选定该值后，相应的，如果前后接收时间差刚好也等于50ms时，那么一个packet-in的处理伴随着一个packet-in的进入，负载率维持稳定，当时间差小于50ms时，队列里报文数量会增加，不过由于接受的过程中，报文的处理也在进行，因此结合二分模型，t值设定在25ms（50/2）较为合适，这里不细展开，只通过一个例子来验证该模型。

|  |
| --- |
| packetInDutyFactor =1, packetInDutyFactor == 0  =packetInDutyFactor+1, curPacketInTime-lastPacketInTime < t  =packetInDutyFactor |

如图3-4所示，假设t时刻传来第一个packet-in报文，此时负载因子设值为1，25ms时，接收到第二个packet-in，负载因子加1，又过25ms，接收到第三个packet-in，负载因子加1，此时总的负载因子为3，由于当第三个packet-in收到时，经过了50ms，此时第一个packet-in的处理过程应当完成，所以需要将负载因子减1，这样一来，最后的负载因子值为2。

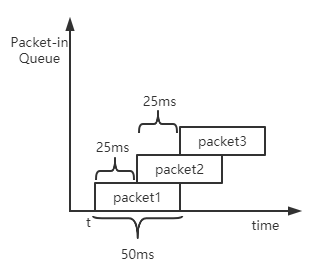


图3-4 packet-in接受/处理时序图

接下来需确定切换到忙碌状态的阈值BUSYTHRESHOLD，该阈值的设值较为朴素，设想如果此时packet-in处理队列中有n个待处理，则总处理时间为50\*n ms，如果网络应用的最大容忍往返时延低于该值就会造成丢包，在本实验中，请求回送报文的等待时间设值为500ms，因此 BUSYTHRESHOLD设值为10（500/50），那么当负载因子超过该值时，控制器切换为忙碌状态，拒绝接受packet-in报文，让当前交换机向其他端口转发数据报。当负载因子低于IDLETHRESHOLD时，切换为空闲状态，可以接受packet-in报文进行处理。未避免频繁在二者之间切换状态，IDLETHRESHOLD值一般小于BUSYTHRESHOLD，本例中设置为5。

忙碌状态下，控制器应当下发一条流表，使得该交换机将接收到的packet-in报文传递到其余交换机上，转发遵循“不返回”与“就近”两条原则，即收到的数据报不向源端口转发，且转发出口应当靠近真实路径，目前，后者处于实验难点，暂未实现。

## 作业分工与进度安排

|  |  |  |
| --- | --- | --- |
| 成员 | 主要工作 | 进度安排 |
| 温雅楠 | 收集攻击原始数据流，提取特征，训练模型（基于BP神经网络的攻击源检测算法） | 实现相关算法，使之能成功运用于单目标攻击的检测与过滤 |
| 武仕沛 | 环境部署，基于packet-in协调转发策略的非法攻击流量防御设计与实现 | 改进转发策略，使交换机能够有效处理来自不同端口的未匹配报文 |
| 王宇翔 | 请求回送报文生成模块与非法攻击流量产生模块的设计与实现 |  |

## 参考资料

[1] https://blog.csdn.net/wangyiyungw/article/details/80537891.

[2] https://blog.csdn.net/AsNeverBefore/article/details/78916645.

[3] https://www.sdnlab.com/2909.html.

[4] https://www.sdnlab.com/experimental-platform/.

[5] https://www.sdnlab.com/sflow-ddos/.

[6] http://blog.chinaunix.net/uid-20556054-id-3164909.html.

[7] https://blog.csdn.net/crystonesc/article/details/68483960

[8] https://www.kancloud.cn/zhaoshuo2016/ovs/262770

[9] https://blog.csdn.net/qq\_45735611/article/details/108649012

[10] 陈飞,毕小红,王晶晶,刘渊.DDoS攻击防御技术发展综述[J].网络与信息安全学报,2017,3(10):16-24.

[11]Mattijs Jonker,Anna Sperotto,Roland van Rijswijk-Deij,Ramin Sadre,Aiko Pras. Measuring the Adoption of DDoS Protection Services[P]. Internet Measurement Conference,2016.

[12]Lukasz Apiecionek, Jacek M.Czerniak, Wojciech T.Dobrosieski. Quality of Services Method as a DDoS Protection Tool. Intelligent Systems 2014 pp 225-234

[13]Elaine Shi, Ion Stoica, David Andersen, Adrian Perrig. OverDoSe: A Generic DDoS Protection Service Using an Overlay Network. 2006-01, CMU-CS-06-114.

[14]RUKAVITSYN A, BORISENKO K, SHOROV A. Self-learningmethod for DDoS detection model in cloud computing[C]//IEEEYoung Researchers in Electrical and Electronic Engineering. 2017:544-547.

[15]WAHAB O A, BENTAHAR J, OTROK H, et al. Optimal loaddistribution for the detection of VM-based DDoS attacks inthe cloud[J]. IEEE Transactions on Services Computing, 1939,(99): 1.

[16]LUA R, YOW K C. Mitigating DDoS attacks with transparent and intelligent fast-flux swarm network[J]. IEEE Network, 2011, 25(4):28-33.

[17]CHEN J, WANG Y, WANG X. On-demand security architecture forcloud computing[J]. Computer, 2012, 45(7): 73-78.