基于 SDN 的 DDoS 攻击防御实现

武仕沛
北京航空航天大学计算机学院

摘　要：分布式拒绝服务攻击（Distributed Denial of Service）即利用大量合法的分布式服务器对目标发送请求，从而导致正常合法用户无法获得服务。传统分布式的网络不具备状态性，难以集中管理，在转发包时，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 攻击流留下可乘之机。而集中式的、基于软件定义网络（SDN）体系架构的网络能够迅速有效地收集南向接口传来的数据，通过自定义控制器模块实现对网络数据的监管和辅助传输。本实验着重研究 SDN 网络下 DDoS 的攻击方的行为，并辅以行之有效的防御措施，使得 SDN 网络变得安全。此次实验攻击方实现链路攻击和单目标攻击两种攻击方式，针对这两种攻击分别在 SDN 网络中部署 packet-in 协调转发和攻击流量检测模块，试验结果表明，在采取防御措施以后，针对 SDN 的 DDoS 攻击影响得到明显抑制。

关键词：分布式拒绝服务攻击；软件定义网络；链路攻击；packet-in 协调转发；攻击流量检测

Implementation of SDN-based DDoS attack and defense

Wu Shipei
Beijing University of Aeronautics and Astronautics

Abstract: Distributed Denial of Service attacks use a large number of legitimate distributed servers to send requests to the target, causing normal legitimate users to be unable to obtain services. Traditional distributed networks are not stateful and difficult to centrally manage. When forwarding packets, they directly use the table look-up method for forwarding without analyzing the data flow, which leaves an opportunity for DDoS attack flows. The centralized, software-defined network (SDN) architecture-based network can quickly and effectively collect data from the southbound interface, and realize the supervision and auxiliary transmission of network data through a custom controller module. This experiment focuses on studying the behavior of the attacker of DDoS under the SDN network, and supplemented by effective defensive measures to make the SDN network safe. In this experiment, the attacker implemented two attack methods: link attack and single-target attack. For these two types of attacks, packet-in coordinated forwarding and attack traffic detection modules were deployed in the SDN network. The test results showed that after taking defensive measures, The impact of DDoS attacks on SDN has been significantly suppressed.

Keywords: distributed denial of service attack; software-defined network; link attack; packet-in coordinated forwarding; attack traffic detection

0 引言

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。

1 相关工作

此次实验由小组三人共同完成，我涉及到的部分为项目整体环境搭建，FloodLight 控制器开发，合法请求流量产生，链路攻击流量产生，packet-in 协调转发策略/模块的设计和实现，并通过测试检验了本人工作的有效性。

2 实验环境搭建

本实验采用 Windows+Linux 双系统联调开发，具体的，Windows 上部署 FloodLight 开发和运行环境，借助于 IDEA 强大的编辑功能，可以方便的进行模块调试，数据通路的查看。而在 Linux 上部署 MiniNet 平台，实现虚拟网络环境的搭建，同时借助于 sFlow 代理工具，实现网络环境中各节点、各端口的流量监测。

双系统联调开发首要保证的就是两平台的联通性，首先是在 Linux 上配置 IP 地址，设置为 192.168.137.181/24，Windows 配置 IP 为 10.4.9.251/24，之后在 mininet 启动过程中，将远端控制器 IP 设置为 Windows 的 IP 地址即可。

FloodLight 作为控制器，启动之初仅充当了一个作为 SDN 网络协调者的黑盒功能，如果要针对本实验的 DDoS 攻击做检测和防御的话，需另行实现检测和处理模块。因此除了使用 FloodLight 外，还应当深入了解该工具源码，学会如何二次开发。FloodLight 源码可在 github.com/floodlight/上下载，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，通过可视化的形式对所管理网络进行展示。通过 maven 打包发布后，通过 jar 命令进行部署。

另在 Linux 环境下，通过 apt-get 方式安装 MiniNet 组网软件，所需的 Java 和 Python 环境以及 sFlow 监测软件均可通过源码包的方式进行安装。

MiniNet 组网软件使用较简单，可通过命令式或图形界面创建拓扑结构，本实验以一个度为 8，深度为 2 的树形网络为例，创建命令为 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。

在 FloodLight 前端界面中，可以图示化的形式展示该网络拓扑，如图 1 所示，图中每个节点都标识了其 mac 地址，方便在 MiniNet 中进行管理。

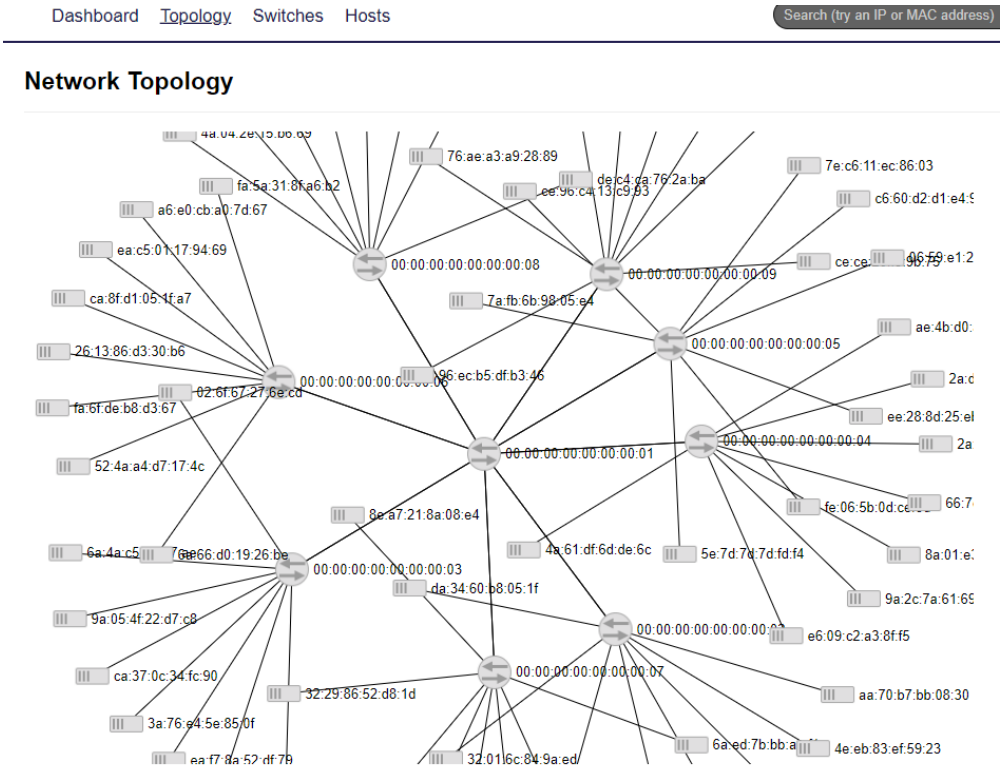

图 1 FloodLight 拓扑结构可视化

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 所示。后续的攻击流量测试就以该环境为基础，通过编写 python 脚步，实现链路带宽攻击。

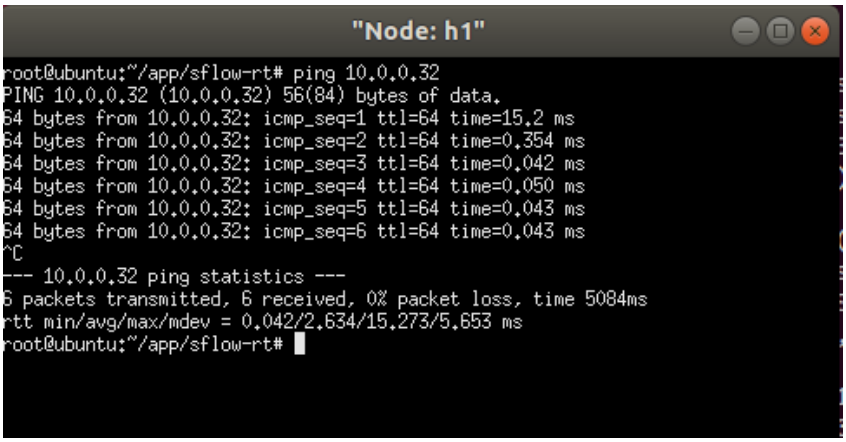

图 2 虚拟主机节点通过 xterm 进行 ping 通测试

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。

后端配置端口默认为 8008，总览界面如图 3 所示。

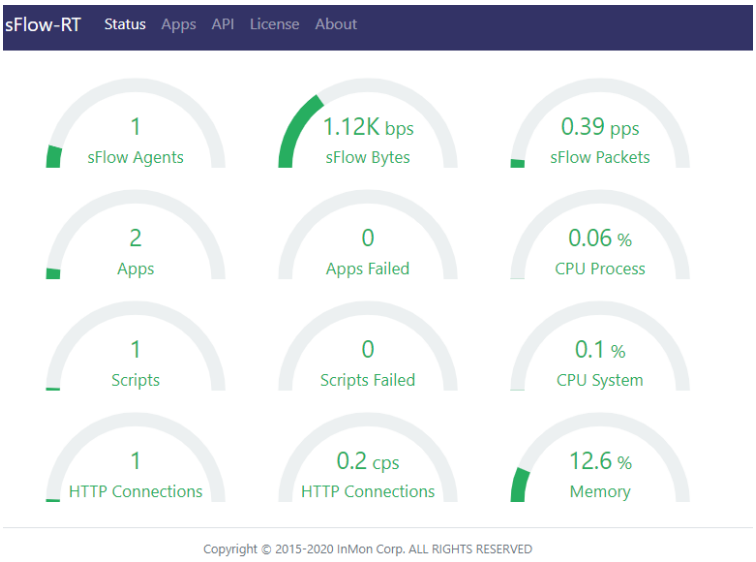

图 3 sFlow 流量监控仪表盘

3 链路带宽攻击

链路带宽攻击在 DDoS 环境下相较于传统网络较为新颖，其攻击的原理不同，但效果相

同，在业界已经有相关论文对该种攻击做过详细阐述，其攻击原理是通过发送大量源、目标IP 等特征伪造的数据包，当 OVS 设备收到该数据包后，查看流表发现无法匹配时，就会向控制器发送 packet-in 报文，请求控制器为该报文计算一条传输路径，当接收到控制器的指示后，就会根据该流表转发该报文，若同一时刻接收到大量的“未知”数据包时，就会因流表缓存溢出从而向控制器发送大量的 packet-in 报文，若超过控制器处理阈值，就会产生时延/丢包，后续真正的合法请求包就会得不到转发，链路攻击示意图如图 4 所示。

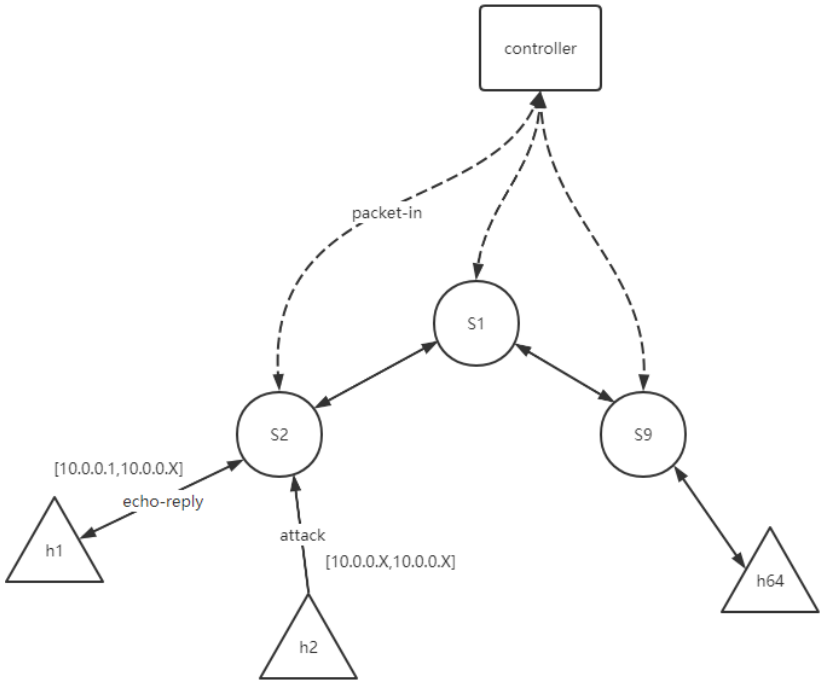

图 4 模拟链路带宽占用示意图

由于该实验设备有限，同一时刻下无法产生足以抑制控制器处理的 packet-in 报文，所以为了方便复现该攻击，人为的在控制器 Forwarding 模块中调低了 packet-in 处理速率，同时减小了 OVS 设备的缓存大小，让其更快产生缓存溢出。

综上通过在 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。

得知攻击原理后，相应的防御机制就比较好实现了，造成链路带宽被占用的原因无非就是 packet-in 大量拥塞，那么解决方案要么从源头上减少 packet-in 的产生，要么将 packet--in 进行分流，相当于做了个负载均衡处理。针对该攻击，由于无法事先得知何种攻击源会产生该攻击报文，因此也就无法采用源头控制的方案了，而是考虑采用后者，后文称该种防御机制为 packet-in 协调转发策略。

4 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忙碌状态的模型，该模型中将控制器接收和处理 packet-in 报文的负载因子定义为 packetInDutyFactor，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,如果接收速率大于处理速率则负载因此上升，否则降低。另外，需要定义控制器的两个状态（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 5 为例，说明负载因子的变化过程以及状态之间的切换过程。

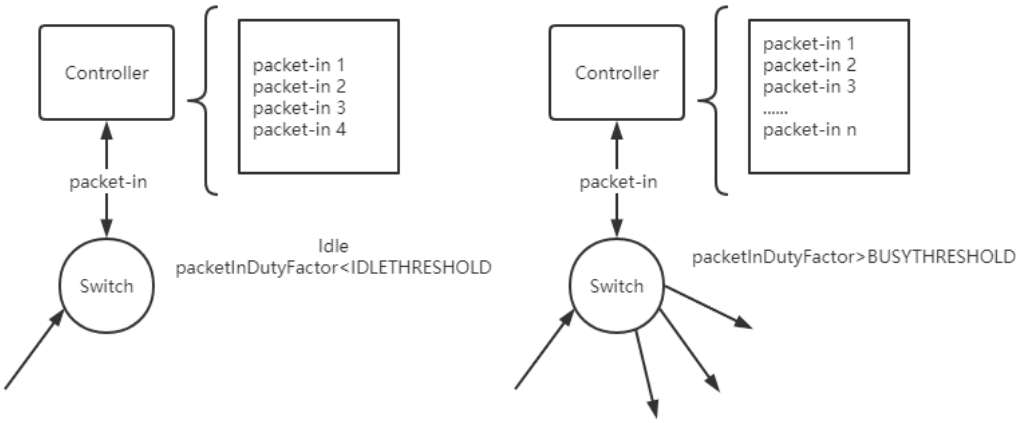

图 5 packet-in 处理导致交换机忙碌/空闲状态的转换示意

首先是 packetInDutyFactor 负载因子的计算过程，大致逻辑如下，控制器启动之初，负载因子初始化为 0，当收到第一个 packet-in 报文时，负载因子增长为 1，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。假如选定处理速率为 50ms，那么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，因此结合统计分布模型，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，如图 6 所示，假设 t 时刻传来第一个 packet-in 报文，此时负载因子增长为 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。算法处理过程如表 1 所示。

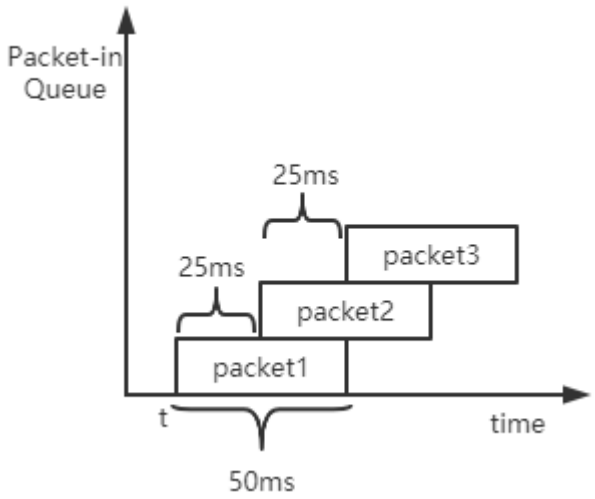

图 6 packet-in 接受/处理时序图

表 1 负载因子增长过程模型

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

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。为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”原则，即收到的数据报不向源端口转发。

5 合法/非法流量生成

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，令目标 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，由 Python 编写，这里仅列举代码关键部分，如代码 1 所示。

代码 1 合法请求流量生成关键代码

```
1    def generateDestinationIP(start, end):
2        first = 10
3        second = 0
4        third = 0
5        ip = ".".join([str(first), str(second), str(third),      str(randrange(start,end))])
```

```
6       return ip

7   for i in xrange(num):
8           #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
9           if(fixed == True):
10              desIp = "10.0.0.64"
11          else:
12              desIp = generateDestinationIP(2, 65)
13          print("get reply from : ip[+"+ desIp +"]")
14          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
```

注意，在针对链路带宽的攻击场景下，攻击会导致网络中部分或全部的链路被攻击，这时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。

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，可合理精细的设置网络中被检验部分，以检测攻击和防御模块的有效性。

非法攻击流量生成模块用于实施对网络链路带宽的攻击。此攻击场景下，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。非法攻击流量生成模块的代码，由 Python 编写，这里仅列举代码关键部分，如代码 2 所示。

代码 2 链路带宽攻击流量生成关键代码

```
        #总共 5 轮攻击，每轮 Dos 攻击 500 次
1       for i in range (1, 5):
2           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
3           for i in xrange(0, 500):
4               if(fixed == True):
5                   desIp = "10.0.0.64"
6               else:
7                   desIp = generateDestinationIP(start, end)
8               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
9               print(repr(packets))
10              sendp(packets, iface = interface.rstrip(), inter = 0.025)
                #发送间隔为 0.025 秒，用于模拟请求泛洪
11          time.sleep (10)
```

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。

6 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。总体处理逻辑为：对于接收到的是 PACKET_IN 报文类型，控制器最终调用了 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。ForwardingBase 实现如代码 3 所示。

代码 3 协调转发模块相关代码

```
1   @Override
2   public Command receive(IOFSwitch sw, OFMessage  msg,
3                                   FloodlightContext cntx) {
4       switch (msg.getType()) {
5           case PACKET_IN://处理 packet-in 报文情况
6               if (packetInDutyFactor == 0) {//如果当前负载率为 0，则更新为 1，
7                   lastPacketInTime = System.currentTimeMillis();
8                   packetInDutyFactor = 1;
9               } 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
10                  packetInDutyFactor++;
```

```
11                  }
12                  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
13                      Curl.execCurl(new String[]{"curl", "-X", "POST", "-d", "@ddos_defend.json",http://127.0.0.1:8080/wm/staticflowentrypusher/json"});
14                  } else {//否则，认为负载率较低，仍按之前方式计算转发路径
15                      IRoutingDecision decision = null;
16                      if (cntx != null) {
17                          decision = IRoutingDecision.rtStore.get(cntx, IRoutingDecision.CONTEXT_DECISION);
18                      }
19                      return this.processPacketInMessage(sw,(OFPacketIn) msg,decision,cntx);
20                  }
21              default:
22                  break;
23          }
24          return Command.CONTINUE;
25      }
```

新增线程，用于监测负载率是否降低到轻负载阈值，如代码 4 所示。

代码 4 负载率降低线程

```
1   new Thread(() -> {
2       while (true) {
3           try {
4               Thread.sleep(50);
5           } catch (InterruptedException e) {
6               e.printStackTrace();
7           }
8           packetInDutyFactor = packetInDutyFactor > 0 ? packetInDutyFactor- 1 : 0;//每隔 packet-in 报文处理时间，负载率减 1
9           if (packetInDutyFactor == IDLETHRESHOLD) {//如果降低到轻负载率，则删除之前下发的流表
10              Curl.execCurl(new String[]{"curl", "-X", "DELETE", "-d", "\"{\\\"name\\\":\\\"no_forward_process\\\"}\"",
                    "http://127.0.0.1:8080/wm/staticflowentrypusher/json"});
11          }
12      }
13  });
```

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。如表 2 所示。

表 2 流表转发规则

```
{
    "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"
}
```

7 实验测试及结果

首先启动 FloodLight 控制程序，java -jar ./target/floodlight.jar，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 7 所示。

```
mininet> nodes
available nodes are:
c0 h1 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h2 h20 h21 h22 h23 h24 h25 h26 h27
 h28 h29 h3 h30 h31 h32 h33 h34 h35 h36 h37 h38 h39 h4 h40 h41 h42 h43 h44 h45 h
46 h47 h48 h49 h5 h50 h51 h52 h53 h54 h55 h56 h57 h58 h59 h6 h60 h61 h62 h63 h64
 h7 h8 h9 s1 s2 s3 s4 s5 s6 s7 s8 s9
```

图 7 nodes 查看虚拟网络节点情况

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 8 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。

```
root@ubuntu:~/app/Packet-Generation# python echo-reply.py -i 0.5 -r 3 -t 1 -f 0

get reply from : ip[+10.0.0.10]
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
get reply from : ip[+10.0.0.43]
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
get reply from : ip[+10.0.0.5]
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
get reply from : ip[+10.0.0.31]
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
get reply from : ip[+10.0.0.43]
Begin emission:
Finished sending 1 packets.
```

图 8 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 9 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。

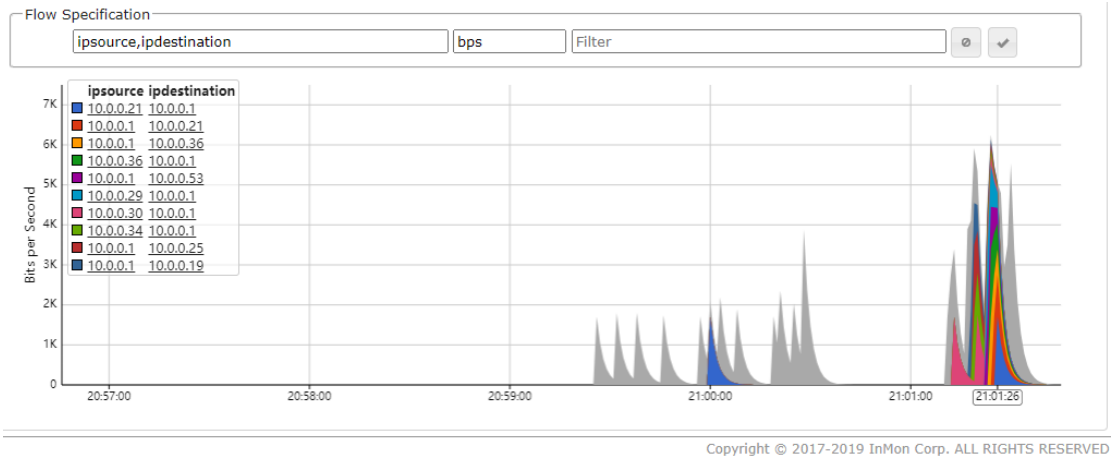

图 9 请求回送流量监测

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 10 所示（真实环境下产生了海量发送日志，图中只截取部分）。

```
Sent 1 packets.
<Ether  type=IPv4 |<IP  frag=0 proto=udp src=10.0.0.19 dst=10.0.0.24 |<UDP  sport=80 dport=1 |>>>
.
Sent 1 packets.
<Ether  type=IPv4 |<IP  frag=0 proto=udp src=10.0.0.13 dst=10.0.0.57 |<UDP  sport=80 dport=1 |>>>
.
Sent 1 packets.
<Ether  type=IPv4 |<IP  frag=0 proto=udp src=10.0.0.11 dst=10.0.0.40 |<UDP  sport=80 dport=1 |>>>
.
Sent 1 packets.
<Ether  type=IPv4 |<IP  frag=0 proto=udp src=10.0.0.17 dst=10.0.0.26 |<UDP  sport=80 dport=1 |>>>
.
Sent 1 packets.
<Ether  type=IPv4 |<IP  frag=0 proto=udp src=10.0.0.50 dst=10.0.0.63 |<UDP  sport=80 dport=1 |>>>
.
Sent 1 packets.
<Ether  type=IPv4 |<IP  frag=0 proto=udp src=10.0.0.63 dst=10.0.0.34 |<UDP  sport=80 dport=1 |>>>
.
Sent 1 packets.
<Ether  type=IPv4 |<IP  frag=0 proto=udp src=10.0.0.32 dst=10.0.0.53 |<UDP  sport=80 dport=1 |>>>
.
Sent 1 packets.
<Ether  type=IPv4 |<IP  frag=0 proto=udp src=10.0.0.4 dst=10.0.0.44 |<UDP  sport=80 dport=1 |>>>
.
Sent 1 packets.
<Ether  type=IPv4 |<IP  frag=0 proto=udp src=10.0.0.36 dst=10.0.0.41 |<UDP  sport=80 dport=1 |>>>
```

图 10 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 11 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 11 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 12 所示，丢包次数减少，但延迟增加。

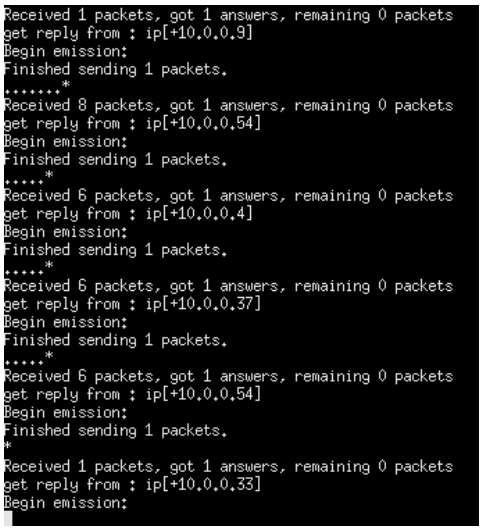

图 12 链路带宽攻击状态下请求回送报文日志（等待时间 1s）

此时再次查看 S2 端口转发流量，如图 13 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。

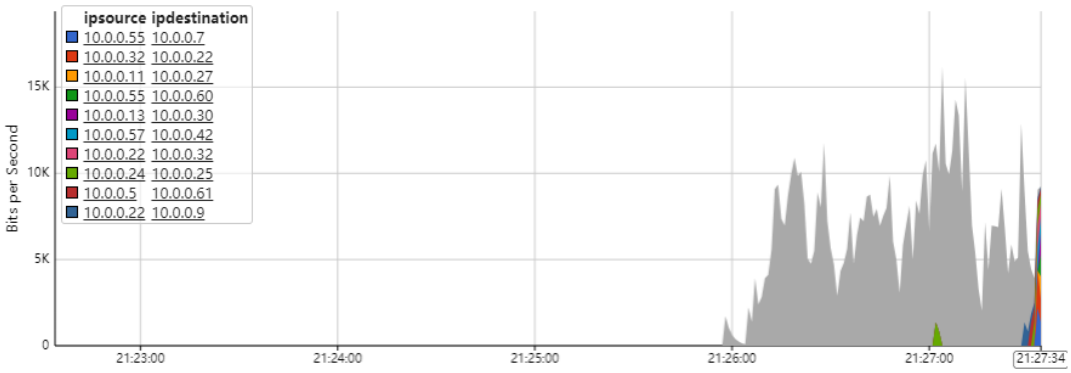

图 13 链路带宽攻击状态下流量监测视图

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，FloodLight 前端显示结果如图 14 所示。

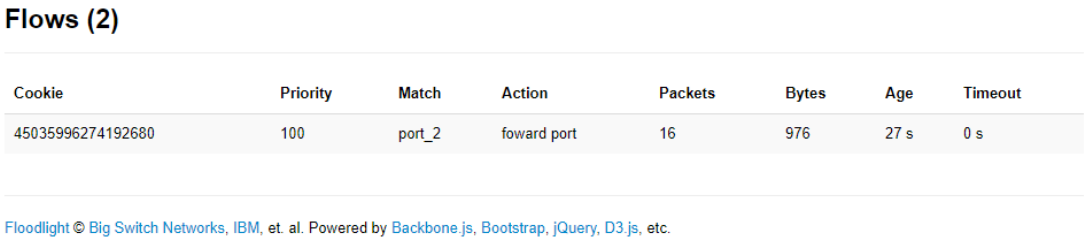

图 14 FloodLight 前端查看 s2 流表

8 结束语

以上就是本次实验中我所做的全部工作，前期通过调研相关研究和理论知识，使我了解到了 SDN 场景下的 DDoS 攻击原理，并对解决算法有了较为系统的学习，并学会了如何针对 FloodLight 进行二次开发，以实现满足项目需求的控制器。虽然实验设备较为简陋，但从宏观上对 SDN 下的链路带宽攻击与防御模型有了一定的认知，这为我以后从事相关研究，特别是 SDN 网络架构下的网络安全提供了不错的基础。

参考文献：

[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.