

Filtering spoofed ip traffic using switching asics

吴欣泽

摘要

在^[1]中，作者设计了一个用于过滤欺骗流量的线速网络系统 NETHCF。NETHCF 利用可编程交换机提供的机会，设计了一种针对欺骗 IP 流量的新型防御方案，具有高效和自适应的特点。一个关键的挑战来自可编程交换机的计算模型和内存资源的限制，并设计了 IP-to-Hop-Count(IP2HC) 映射表以高效地使用内存。

关键词：可编程交换机；欺骗 IP 流量；跳数过滤

1 引言

欺骗 IP 仍然是互联网中的一大威胁，尽管攻击者可以伪造 IP 报中的任何字段，但很难伪造出正确的跳数，先前的工作有提出过在终端进行基于跳数的过滤（HCF），可编程数据平面的出现允许将这种智能过滤卸载到网络。^[1]提出了 NetHCF，通过在可编程交换机中设置 IP2HC 表，监控连接的状态，它能够为合法流量提供带宽保护，为终端服务器节省计算和存储资源，不需要采样就可以更新 IP2HC 表，并以线速率进行欺骗包检测和过滤。

相较于传统的部署在终端的 HCF，NetHCF 面临着以下 2 个挑战：1、交换机的 SRAM 空间有限，如何在较低假阳率（附带损害）的情况下存储 IP2HC 表；2、有限映射（无法存储全部映射）。为此本文使用哈希来存储部分流行映射，并提出了 NetHCF 的两种运行状态以决定收到一个错误跳数的报文时是选择丢弃还是学习。

首先 IP2CH 表是一个 32 位图，既然基于哈希，那么冲突不可避免，其次表记录的是合法的连接映射，这些映射是需要不断更新的，原文作者提出了一种更新表的思路。作者已开源 BMv2 版本的 P4 源码，但是这份源码仅使用了一个哈希函数，为了进一步减少冲突和假阳率，我们可以用不同的哈希函数对 IP 地址进行多次哈希，另外源码没有对表的更新操作做实现。本文主要对上述两点进行了复现和改进

2 相关工作

在本节中，我们将介绍更多有关欺骗数据包过滤技术和 HCF 方案的背景知识，描述现有 HCF 设计的缺点，并讨论可编程交换机为何提供新的机会。

2.1 HCF 背景

源地址欺骗是困扰互联网的最严重问题之一。现有的防御系统可分为基于路由器的防御系统和基于主机的防御系统。基于路由器的方法在路由器内部安装防御机制，以追踪攻击源^{[2]、[3]、[4]}，或以协调的方式检测和阻止攻击流量^{[5]、[6]、[7]}。然而，这些解决方案不仅需要修改路由器，还需要在路由器甚至网络之间进行协调。相比之下，基于主机的方法部署起来要容易得多。边缘网络、小型互联网服务提供商、数据中心或企业等终端系统也有针对欺骗流量部署防御机制的自然动机。基于主机的方法使用复杂的来源识别方案^{[8]、[9]、[10]}，或减少每个请求^{[11]、[12]}的资源消耗以减轻攻击。HCF^{[13]、[14]}是一种简单但有效的基于主机的解决方案。它可以在互联网服务器上验证传入的 IP 分组，而无需任何密码操作

或路由器修改，使其变得轻量级和实用。HCF 还有效地打破了攻击者和受害者之间的成本不对称——受害者可以轻松地为防御建立合法的 IP2HC 映射，而攻击者无法轻松获得任意 IP 地址及其到受害者的跳数之间的映射。具体地说，从受害者的角度来看，他们可以通过从目的地的初始 TTL 中减去最终 TTL 来轻松推断跳数信息。由于这些跳数由 Internet 路由基础设施确定，这也使得攻击者很难在其攻击流量中使用正确的跳数值。

2.2 可编程交换机

软件定义网络 (SDN) 的最新发展导致了可编程交换 ASIC^[15]和特定于域的语言 (例如，P4^[16])，以将网络可编程性从控制平面扩展到数据平面。与传统的固定功能交换机相比，新兴的可编程交换机在不牺牲性能的情况下提供了硬件可编程性。它们的功耗和价格也与普通交换机相似。新的硬件为克服传统 HCF 设计的缺点提供了独特的机会。

3 本文方法

3.1 本文方法概述

源代码的数据平面和控制平面是分离的。NetHCF 有两种模式，一种是学习模式，一种是过滤模式。在学习模式下 NetHCF 会记录跳数不正常的数据包，如果是已经建立 TCP 连接的源 IP 的数据包，NetHCF 还会学习这个跳数并更新到 IP2HC 表中；在过滤模式下所有跳数不正常的数据包或没有连接状态的数据包都将被过滤。NetHCF 的数据平面主要是 P4 代码，P4 语言可以对交换机寄存器读写，对数据包进行监控和过滤，实现了一个基础的 NetHCF；控制平面也可以读写寄存器，可以更改 NetHCF 的模式。

3.2 数据平面

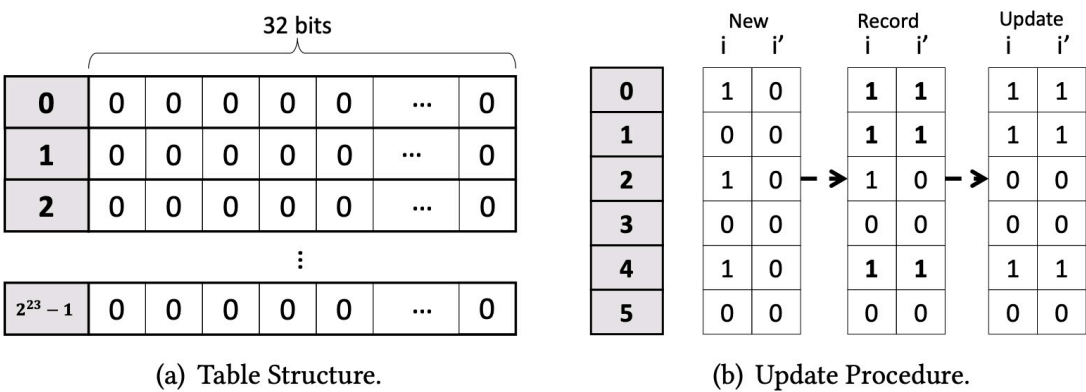


Figure 1: IP2HC Table.

图 1: IP2HC 表样式^[1]

源 P4 代码主要的工作量是在 ingress 上，其主要负责监控 TCP 连接，并保存未完成三次握手的连接的状态，对于已完成三次握手的源 IP 请求，如果跳数不正常且 NetHCF 在学习模式，那么它将记录新的跳数到 IP2HC 表中；否则数据包会被过滤。IP2HC 表如图 1a 所示，有关表的介绍可以在原文^[1]中查阅。值得注意的是，原文有提到因为数据连接是不断变化的，这张表也是需要不断更新的，但是由于这张表没有时间维度，那么如何更新表的内容成为了一个问题，原文提出用多一倍的存储空间用作缓存表，已记录过去一段时间访问过的连接，每隔一段时间将缓存表覆盖到原表中并重置缓存表，如

图 1b 所示。关于这个更新源代码并没有实现，在这篇复现论文中我对他进行了实现（虽然并没有完全实现，这会在章节 4 中解释）。此外 IP2HC 表使用的是哈希记录，既然使用哈希那么势必会出现冲突，原文提出的方案是使用多个哈希函数取得多个哈希索引来改善冲突，但是源 P4 代码中仅使用一个哈希函数，因此复现还增加了一次哈希，并优化了部分 P4 代码。

3.3 控制平面

源代码的控制平面主要实现的功能是：每隔一段时间，读取异常包计数器，并根据得到的值判断是否要改变 NetHCf 模式，然后清空计数器并重复以上操作。此外在上一节中提到的更新操作，有一部分代码也是写在控制平面的。更新操作也是一个周期操作，每过一段时间将缓存表的值覆盖到原表中并重置寄存器。

4 复现细节

4.1 与已有开源代码对比



图 2: 源代码 topo

源代码是建立在如图 2 所示的 topo 结构上的。首先为了测试源代码的可用性，本人使用 python 语言写了一个发包和收包的脚本（在 github 上¹是 send.py 和 reveive.py）。这两个脚本利用 scapy 包实现了 TCP 三次握手，当然稍作更改也可以实现在短时间内快速地发送 UDP 数据包来模拟攻击。

```
control ingress {
    // Get basic information of switch and tcp session
    apply(hcf_check_table); //判断当前状态, 默认1检查, 端口2发出的不查
    apply(session_check_table); //根据五元组给session状态及seq
    if (meta.tcp_session_state == 1) {
        // The connection is waiting to be established
        if (tcp_ackno == meta.tcp_session_seq + 1) {
            // Legal connection, so real hop count to be stored
            apply(hc_compute_table_copy); //计算跳数
            apply(session_complete_table); //session状态置0, ip2hc表中 (行号为源ip哈希值, 列号为跳数) 的位置更新
        }
        else {
            // Illegal connection attempt
            apply(packet_abnormal_table);
        }
    }
    else if (meta.tcp_session_state == 0) {
        // TCP session has been established or not
        if (tcp_syn == 1 and tcp_ack == 1) {
            // A client is attempting to connect to the server
            apply(session_init_table);
        }
        else if (meta.is_inspected == 1) {
            // Other packets. Anyway, sample it first
            apply(packet_sample_table);
            if (meta.sample_value == 0 or meta.hcf_state == 1) {
                // The packet is sampled or the switch is in filtering state
                // Compute packet's hop count and refer to its origin hop count
                apply(hc_compute_table);
                apply(hc_inspect_table);
                if (((meta.ip_to_hc_bitmap >> meta.hop_count) & 1) == 0) {
                    // Diverse Hop Count. The reason may be two initial TTL pairs
                    // So recompute hop count for those packets
                    apply(hc_compute_twice_table);
                    if (((meta.ip_to_hc_bitmap >> meta.hop_count) & 1) == 0) {
                        // It must be abnormal packet
                        apply(hc_abnormal_table);
                    }
                }
                else {
                    // It is normal
                    apply(packet_normal_table);
                }
            }
            else {
                // It is normal
                apply(packet_normal_table);
            }
        }
        else {
            // Do nothing to these packets
            apply(packet_normal_table);
        }
    }
}
// Drop abnormal packets and forward normal packets in layer two
apply(l2_forward_table);
```

```
control ingress {
    // Get basic information of switch and tcp session
    apply(hcf_check_table); //判断当前状态, 默认1检查, 端口2发出的不查
    apply(session_check_table); //根据五元组给session状态及seq
    if (meta.tcp_session_state == 1) {
        // The connection is waiting to be established
        if (tcp_ackno == meta.tcp_session_seq + 1) {
            // Legal connection, so real hop count to be stored
            apply(hc_compute_table_copy); //计算跳数
            apply(session_complete_table); //session状态置0, ip2hc表中 (行号为源ip哈希值, 列号为跳数) 的位置更新
        }
        else {
            // Illegal connection attempt
            apply(packet_abnormal_table);
        }
    }
    else if (meta.tcp_session_state == 0) {
        // TCP session has been established or not
        if (tcp_syn == 1 and tcp_ack == 1) {
            // A client is attempting to connect to the server
            apply(session_init_table);
        }
        else if (meta.is_inspected == 1) {
            // Other packets. Anyway, sample it first
            apply(packet_sample_table);
            if (meta.sample_value == 0 or meta.hcf_state == 1) {
                // The packet is sampled or the switch is in filtering state
                // Compute packet's hop count and refer to its origin hop count
                apply(hc_compute_table);
                apply(hc_inspect_table);
                if (((meta.ip_to_hc_bitmap >> meta.hop_count) & 1) == 1 and ((meta.ip_to_hc_bitmap2 >> meta.hop_count) & 1) == 1) {
                    apply(hc_normal_table);
                }
                else {
                    apply(hc_compute_twice_table);
                    if (((meta.ip_to_hc_bitmap >> meta.hop_count) & 1) == 1 and ((meta.ip_to_hc_bitmap2 >> meta.hop_count) & 1) == 1) {
                        apply(packet_normal_table);
                    }
                }
            }
            else {
                // Do nothing to these packets
                apply(packet_normal_table);
            }
        }
        else {
            // Do nothing to these packets
            apply(packet_normal_table);
        }
    }
}
// Drop abnormal packets and forward normal packets in layer two
apply(l2_forward_table);
```

图 3: 数据平面代码对比

¹复现后的代码已上传到 <https://github.com/szu-advtech/AdvTech>

```

field_list_calculation ipsrc_map_hash {
    input {
        ipsrc_hash_fields;
    }
    algorithm : crc16;
    output_width : IP_T0_HC_INDEX_BITS;
}

```

```

//两次哈希
field_list_calculation ipsrc_map_hash2 {
    input {
        ipsrc_hash_fields;
    }
    algorithm : crc32;
    output_width : IP_T0_HC_INDEX_BITS;
}

```

图 4: 两次哈希改进

在数据平面上的改动主要是 IP2HC 表更新操作，以及 P4 中 ingress 代码部分的优化，具体对比如图 3 和图 4 所示。其中图 3 左侧为源码，右侧为复现代码，红框所示的改进主要是考虑到两次哈希后 IP2HC 表中会对应生成两个映射值。源代码是只有比特位为 0 的才是不正常的，其他都是正常；改进代码是只有为 1 的才是正常的，其他都是不正常的。试想一下这么一个情况，两个来自不同源的合法数据包且具有相同的跳数，分别被两次哈希后，得到的 4 个哈希值中，有两个是相同的，此时源代码的判断比特位为 0 的方案是行得通的。但是在一段时间过后，其中一个源断开了连接，但是另一个源还在继续，那么这三个哈希值有一个是 0，按照源代码的思路，断开连接的源如果又发起了第一次握手，他对应的比特位是 0 和 1，会被认为是正常的包（事实上他还没建立 TCP 连接），而改进后只有两个比特位都是 1 才会被认为是正常的包。图 4 中左侧是源代码，右侧是改进代码，红框中为改进部分，增加了一次 crc32 哈希。其次数据平面实现更新操作的代码主要是设置一些寄存器，只需要在适当的时机（比如更新表时同时更新缓存表）读写寄存器就行，这里不做展示。

```

#双倍寄存器空间来表示时间维度上的表更新, 这里实现不完整, readme中有说明了
def update_register(nums): #nums是修改过的比特位数量
    for i in range(0, nums): #理论上应该是循环2^23次, 这里为了测试功能, 而没有完全实现
        index = read_reg_record_ip_to_hc(i) #修改过的索引
        read_cmd = 'echo "register_read ip_to_hc2 ' + str(index) + '" | /home/myp4/Downloads/Anti-spoof/
p4_environment/behavioral-model/targets/simple_switch/sswitch_CLI hop_count.json 22223'
        result1 = os.popen(read_cmd).read()
        if "Done" in result1:
            pass
        else:
            print "Error: Can't read register ip_to_hc2 !\n"
            print error_hint_str
            return -1

        write_cmd = 'echo "register_write ip_to_hc ' + str(index) + '" | /home/myp4/Downloads/Anti-spoof/
p4_environment/behavioral-model/targets/simple_switch/sswitch_CLI hop_count.json 22223'
        result2 = os.popen(write_cmd).read()
        if "Done" in result2:
            pass
        else:
            print "Error: Can't write register ip_to_hc !\n"
            print error_hint_str
            return -1
    print "Debug: update successfully!"
    return 0

```

图 5: 控制平面更新操作改进

在控制平面上源代码使用 python 写了一个控制 NetHCF 的 python 脚本, 在此基础上复现写了另一个脚本来实现 IP2HC 表的更新操作, 但由于实现的逻辑是通过 python 到 cmd 到 CLI, 这是一个用 bmv2 实现的软件交换机, 因此代码运行效率并不高, 如图 5 所示, 每当缓存表覆盖原表时, 需要操作 32×2^{23} 这么多个比特位, 然而现实中交换机是在不断处理信息的, 控制平面的操作必须在极快的时间内完成, 否则就需要考虑 runtime 策略, 即在操作寄存器时新来的数据包如何处理? 很显然让他等待是不可能的, 也正因如此这个改进也可以说是没有实现, 因为完全实现它是需要配合硬件的, 甚至可能还需要更底层的代码。在图 5 中, 我尝试只修改最近访问过的比特位, 其他位不操作 (这种情况下的操作速度是够的, 但是这也并没有完全实现, 因为已经断开的连接位并没有用 0 覆盖它, 但这已经可以证明这个控制平面代码是可以跑通的)。在改进过程中, 仅仅操作 100 个比特位就能感受到秒级的延迟了。

4.2 实验环境搭建

本实验使用 VMware Workstation 16 Pro 虚拟机搭载 ubuntu16.04 LTS 系统, 分配 ram 4g, rom 30g (已使用 19.5g)。另外需要在 github 下载 bmv2 和 p4c 编译器作为 P4 源码环境, 可参考教程², python 环境为 python2.7.12 (python2 即可)。如果有需要 ova 镜像, 请邮箱联系³。

4.3 界面分析与使用说明

可参考上传代码的 README

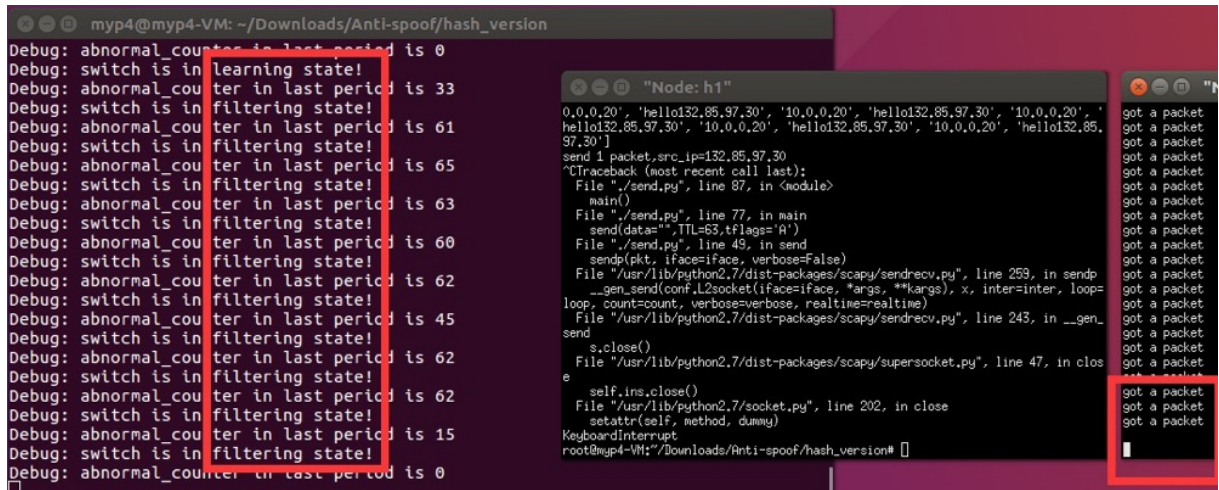
4.4 创新点

本次复现在源代码的基础上, 弥补了源代码中 IP2HC 表的更新操作的空缺, 使用了双哈希函数降低冲突概率, 同时还补充了三次握手脚本, 原文思路基本实现。

²P4 快速上手 <https://www.sdnlab.com/22512.html>

³本人邮箱 wuxinze2022@email.szu.edu.cn

5 实验结果分析



```
myp4@myp4-VM: ~/Downloads/Anti-spoof/hash_version
Debug: abnormal_counter in last period is 0
Debug: switch is in learning state!
Debug: abnormal_counter in last period is 33
Debug: switch is in filtering state!
Debug: abnormal_counter in last period is 61
Debug: switch is in filtering state!
Debug: abnormal_counter in last period is 65
Debug: switch is in filtering state!
Debug: abnormal_counter in last period is 63
Debug: switch is in filtering state!
Debug: abnormal_counter in last period is 60
Debug: switch is in filtering state!
Debug: abnormal_counter in last period is 62
Debug: switch is in filtering state!
Debug: abnormal_counter in last period is 45
Debug: switch is in filtering state!
Debug: abnormal_counter in last period is 62
Debug: switch is in filtering state!
Debug: abnormal_counter in last period is 62
Debug: switch is in filtering state!
Debug: abnormal_counter in last period is 15
Debug: switch is in filtering state!
Debug: abnormal_counter in last period is 0

Node: h1
0.0.0.20: 'hello132.85.97.30', '10.0.0.20', 'hello132.85.97.30', '10.0.0.20', '
hello132.85.97.30', '10.0.0.20', 'hello132.85.97.30', '10.0.0.20', 'hello132.85.
97.30']
send 1 packet,src_ip=132.85.97.30
^CTraceback (most recent call last):
  File "./send.py", line 87, in <module>
    main()
  File "./send.py", line 77, in main
    send(data="",ttl=63,tflags='A')
  File "./send.py", line 49, in send
    sendp(pkt,iface=iface,verbose=False)
  File "/usr/lib/python2.7/dist-packages/scapy/sendrecv.py", line 259, in sendp
    _gen_send(conf,sock(socket(iface=iface,*args,**kargs)),x,inter=inter,loop=
loop,countcount,verbose=verbose,realtime=realtime)
  File "/usr/lib/python2.7/dist-packages/scapy/sendrecv.py", line 243, in _gen_
send
    s.close()
  File "/usr/lib/python2.7/dist-packages/scapy/supersocket.py", line 47, in clos
e
    self.ins.close()
  File "/usr/lib/python2.7/socket.py", line 202, in close
    setattr(self,method,dummy)
KeyboardInterrupt
root@myp4-VM:~/Downloads/Anti-spoof/hash_version#
```

图 6: 控制平面更新操作改进

如图 6 所示，图中模拟的是 h1 伪造大量随机 IP 数据包攻击 h2 的场景，可以发现 NetHCF 的状态首先由学习模式变成了过滤模式，其次是不正常包的计数，同时在进入过滤模式后 h2 也收不到数据包了（这点不太容易看出来，我敲击了一个回车空行，空行没有被顶上去，说明没继续收到数据包，可以修改代码让他更明显一点）。包括双哈希、更新操作都可以在控制层面验证，本人已在 `control_update.py` 中预留 `test()` 函数，在 P4 代码中设置一下寄存器即可验证。本次复现所做的改进从理论上分析是只有正收益的，加上原文只是一个海报，对实验细节没有过多的描述，因此本复现也没有做详细的结果分析，比如误检率、假阳率之类的。

6 总结与展望

这次复现，虽然在全面实现 NetHCF 的过程中碰到了一些困难，比如学习 P4 语言时的坎坷，以及更新操作没有完全实现（主要是因为软件交换机难以在线速度时间内完成亿级别的比特操作），但这使我对 P4 语言、可编程交换机有了深一步的了解。未来会考虑用 P4 和可编程交换机实现更复杂的过滤系统，比如在不对称路径下如何实现 HCF 等。如果有 P4 语言方面的交流，欢迎邮箱讨论。

参考文献

- [1] BAI J, BI J, ZHANG M, et al. Filtering spoofed ip traffic using switching asics[C]// Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos. 2018: 51-53.
- [2] LI J, SUNG M, XU J, et al. Large-scale IP traceback in high-speed Internet: Practical techniques and theoretical foundation[C]// IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004. 2004: 115-129.
- [3] SAVAGE S, WETHERALL D, KARLIN A, et al. Practical network support for IP traceback[C]// Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication. 2000: 295-306.
- [4] SNOEREN A C, PARTRIDGE C, SANCHEZ L A, et al. Hash-based IP traceback[J]. ACM SIGCOMM Computer Communication Review, 2001, 31(4): 3-14.

- [5] IOANNIDIS J, BELLOVIN S M. Implementing pushback: Router-based defense against DDoS attacks [J]., 2002.
- [6] KEROMYTIS A D, MISRA V, RUBENSTEIN D. SOS: Secure overlay services[J]. ACM SIGCOMM Computer Communication Review, 2002, 32(4): 61-72.
- [7] LI J, MIRKOVIC J, WANG M, et al. SAVE: Source address validity enforcement protocol[C]// Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies: vol. 3. 2002: 1557-1566.
- [8] BANGA G, DRUSCHEL P, MOGUL J C. Resource containers: A new facility for resource management in server systems[C]// OSDI: vol. 99. 1999: 45-58.
- [9] QIE X, PANG R, PETERSON L. Defensive Programming: Using an Annotation Toolkit to Build {DoS-Resistant} Software[C]// 5th Symposium on Operating Systems Design and Implementation (OSDI 02). 2002.
- [10] SPATSCHECK O, PETERSON L L. Defending against denial of service attacks in Scout[C]// OSDI: vol. 99. 1999: 59-72.
- [11] JUELS A. Client puzzles: A cryptographic countermeasure against connection depletion attacks[C]// Proc. Networks and Distributed System Security Symposium (NDSS), 1999. 1999.
- [12] WANG X, REITER M K. Defending against denial-of-service attacks with puzzle auctions[C]// 2003 Symposium on Security and Privacy, 2003. 2003: 78-92.
- [13] JIN C, WANG H, SHIN K G. Hop-count filtering: an effective defense against spoofed DDoS traffic[C]// Proceedings of the 10th ACM conference on Computer and communications security. 2003: 30-41.
- [14] WANG H, JIN C, SHIN K G. Defense against spoofed IP traffic using hop-count filtering[J]. IEEE/ACM Transactions on networking, 2007, 15(1): 40-53.
- [15] BOSSHART P, GIBB G, KIM H S, et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN[J]. ACM SIGCOMM Computer Communication Review, 2013, 43(4): 99-110.
- [16] BOSSHART P, DALY D, GIBB G, et al. P4: Programming protocol-independent packet processors[J]. ACM SIGCOMM Computer Communication Review, 2014, 44(3): 87-95.