第三题：恶意网站防护

目录

第一章 背景介绍

一、实验背景

互联网中存在恶意网站，严重威胁正常上网，例如：www.1cbc.com，仿冒www.icbc.com，意图引诱用户登录看起来及其真实的假冒网站，如果访问可能被套取网银密码，获取用户在该网站上输入的个人敏感信息，造成资金损失。越来越多的研究者认为恶意网站给互联网用户带来潜在的经济损失和可能的信息安全威胁，而且用户安全防范意识的薄弱更促进了恶意网站极大的发展。因此，如何有效进行恶意网站防护具有非常重要的意义。

传统恶意网站防护一般采用在 PC 客户端上安装杀毒软件或安全浏览器，或者在网络出口端安装流量审计软件，客户端安装软件可能不是所有用户都会安装或使用，网络出口流量审计一般是事后审计，可能用户已经造成了损失。

因此，如何建立一个不需要在客户端安装特定软件而且能够事前防护，并且具有高安全性、准确性和稳定性的恶意网站防护系统至关重要。

二、实验目的

搭建简单网络，本题实验利用 SDN 技术设计一种不需要在客户端安装特定软件，且能够事前防护，即在用户访问恶意网站前进行阻止，防止用户数据传输到恶意网站的系统。不仅如此，我们还在恶意网站防护方案基本功能基础上，增加了白名单检测模块、名单修改模块以及可疑域名智能检测模块，使系统有更高的准确性、稳定性以及适用性，满足用户个性化定制的要求。

三、实验环境搭建

如图 1-1 所示，我们会创建一个基于 OpenFlow Switch 的网络实际平台。三台装有 NetFPGA[3, 4]的主机 PC-A、PC-B、PC-C 实现 OpenFlow Switch 的功能，PC-D 实现 OpenFlow Switch Controller，图中的 PC-0，PC-1 是客户端 PC 机。

实验网络采用两层网络架构园区网模型，接入+汇聚设备。PC-A 与 PC-C 作为接入层交换机接入 PC 客户端，PC-B 作为汇聚交换机接入 Internet，其中 PC-A、PC-B、PC-C 作为 OpenFlow Switch 分别与 PC-D 相连。表 1-1 为网络实验环境的硬件配置与软件环境以及参数配置。

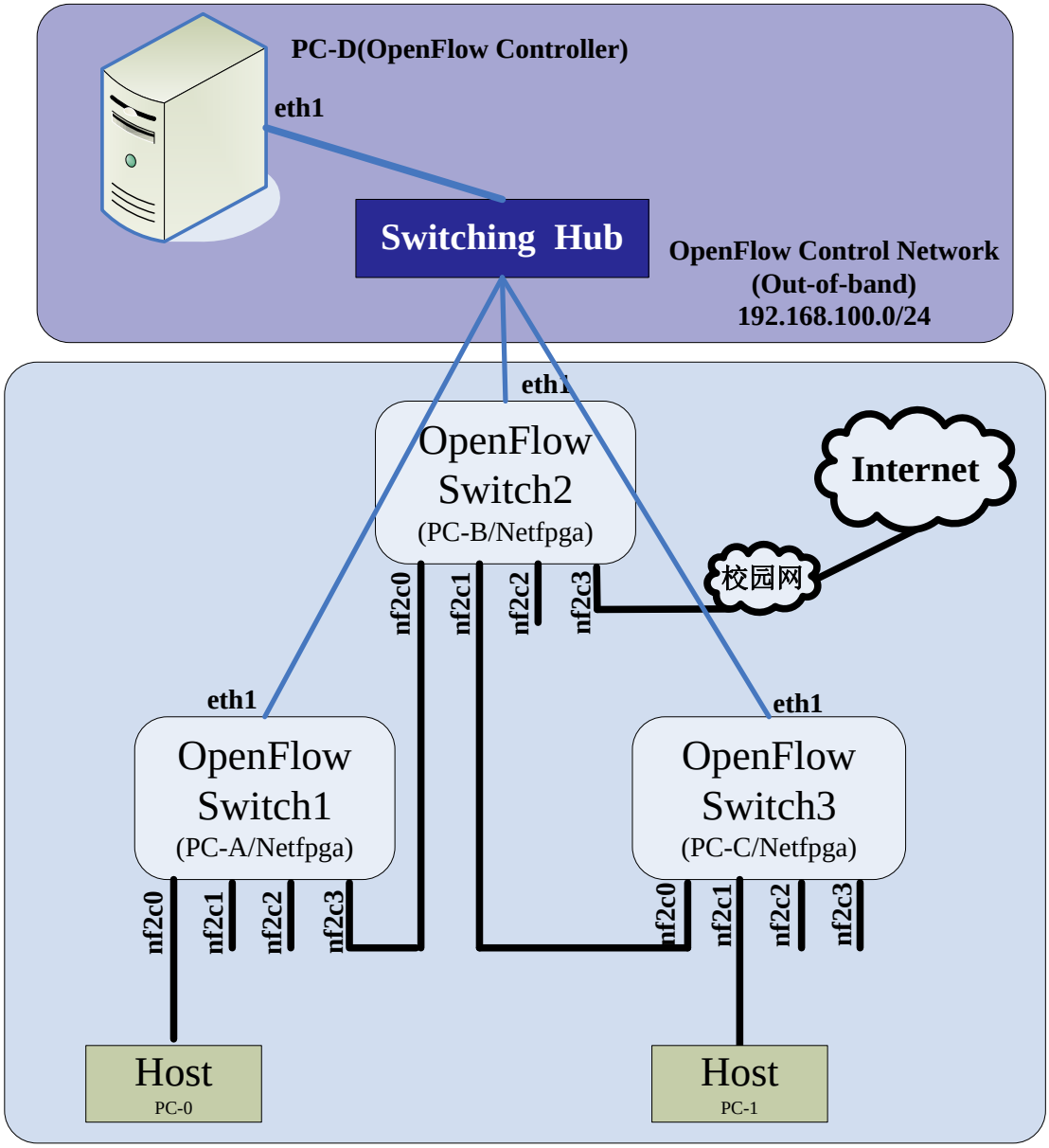

图 1-1 基于 NetFpga 的 Openflow Switch 网络物理实验平台

表 1-1 网络实验环境的硬件配置与软件环境以及参数配置

物理主机	角色	操作系统	内存	IP
PC-A	OpenFlow Switch 1 With Netfpga	Centos 5.6 32 位	2G	Eth1:192.168.100.11/24 MAC:40:16:9F:F3:44:C0
PC-B	OpenFlow Switch 2 With Netfpga	Centos 5.6 32 位	2G	Eth1:192.168.100.22/24 MAC:00:1F:D0:92:25:5B
PC-C	OpenFlow Switch 3 With Netfpga	Centos 5.6 32 位	2G	Eth1:192.168.100.44/24 MAC:00:19:DB:47:BD:1E
PC-D	OpenFlow Controller （采用 POX[1]）	Ubuntu 10.04 32 位	2G	Eth1:192.168.100.100/24 MAC: 00:0C:29:A9:2B:4E
PC-0	HostA	Fedoral 13 32 位	2G	MAC:68:05:CA:08:1F:C4 Eth1:172.18.216.49 Netmask:255.255.255.0 Gateway:172.18.216.254 DNS:222.200.160.1
PC-1	HostB	Fedoral 13 32 位	2G	MAC:14:CF:92:F9:FF:6B Eth1:172.18.216.42 Netmask:255.255.255.0 Gateway:172.18.216.254 DNS:222.200.160.1

[1]本实验用的是 POX+POXDesk。POXDesk 是 POX 的一个组件，增加了 web UI，新增显示 switch 和 host 组成的网络拓扑、显示 switch 的流表等功能。下载地址：https://github.com/09zwcbupt/undergrad_thesis

第二章 实验方案设计

整个恶意网站防护的方案我们将会在完成基本功能（用 SDN 技术设计一种不需要在客户端安装特定软件，且能够事前防护，即在用户访问恶意网站前进行阻止，防止用户数据传输到恶意网站）的基础上，增加白名单检测模块、名单修改模块以及可疑域名智能检测模块。

2.1 恶意网站防护方案基本功能的实现

2.1.1 方案基本功能系统框架及工作流程

（一）方案基本功能系统框架

恶意网站防护方案基本功能系统框架如图 2-1 所示：

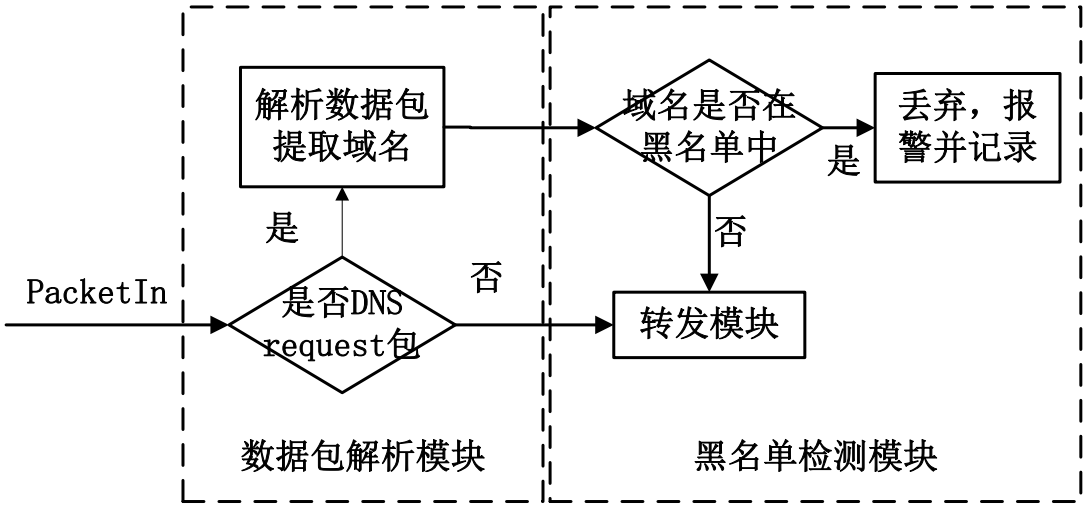

图 2-1 恶意网站防护方案基本功能系统框架

1．数据包解析模块

POX 对收到的 PacketIn 消息进行解析，若发现是 DNS request 包，则提取出域名，并转发到黑名单检测模块；若不是将数据包送到转发模块。

2．转发模块

在 POX 自带的 l2_learning 模块[5]的基础上进行修改，功能与原理同 l2_learning。

3．黑名单检测模块

将恶意域名列表读入，逐条域名存放到 BF(BloomFilter[6, 7]中，形成 BL_BF(BlackList BF)。对送来的域名进行多个哈希运算，判断其是否在 BL_BF 中。若在，POX 则将数据包丢弃，在命令行里显示报警信息，并将信息记录到 log 日志；若不在，POX 则将域名送入转发模块。

数据来源：

恶意网站实验室的恶意网站屏蔽列表 http://www.mwsl.org.cn/hosts/hosts 表中收录 13923 个恶意域名。

（二）方案基本功能工作流程

恶意网站防护方案基本功能工作流程如图 2-2 所示：

1．终端发起 DNS 请求

2．OpenFlow Switch 收到 DNS 请求，发起带有查询信息的报文封装成 Packet_in 的消息给 SDN 控制器。

3．SDN 控制器根据 OpenFlow Switch 上发的报文进行包解析，并与本地的黑名单进行匹配，对黑名单内的域名的包进行丢弃，对黑名单外的域名进行动态流表分发，使之接入校园网，经过网关接入 Internet，实现 DNS 请求响应。

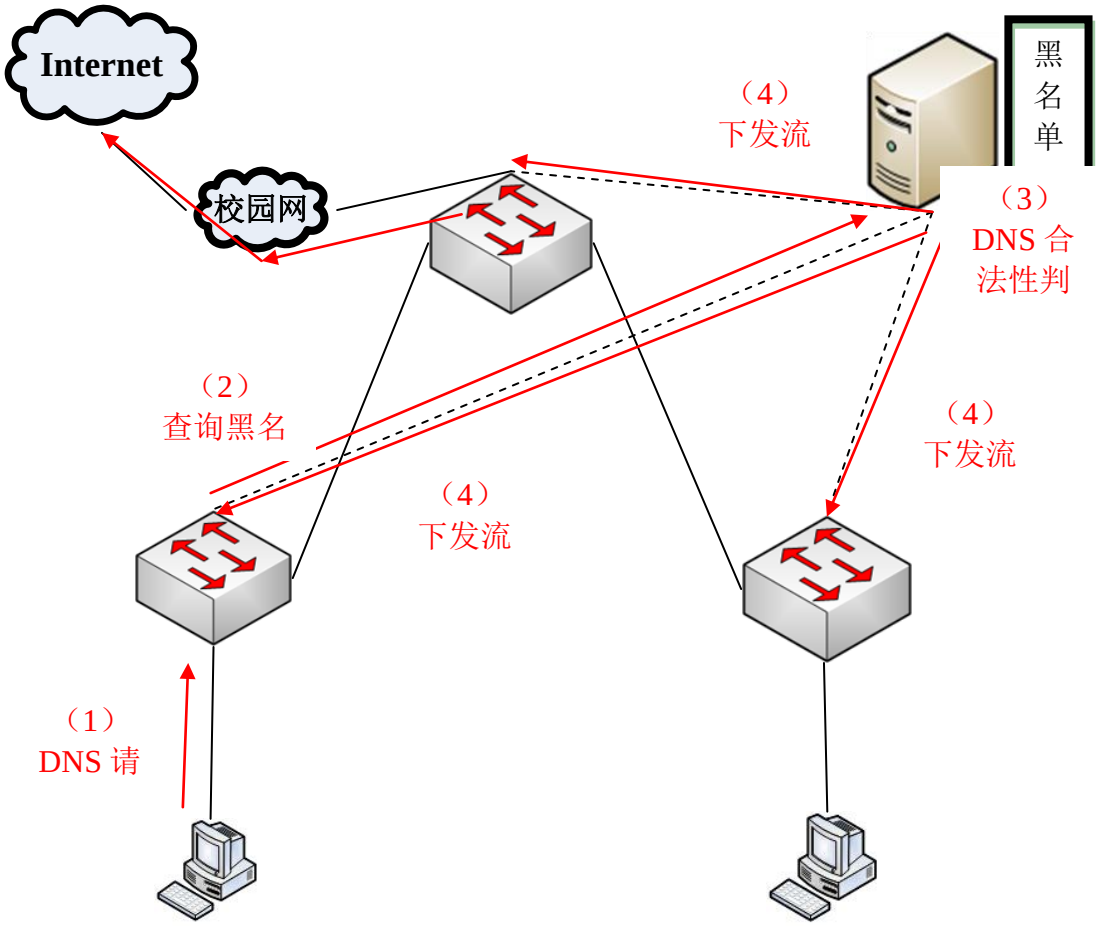

图 2-2 恶意网站防护方案基本功能工作流程

2.1.3 方案基本功能的关键难点

恶意网站防护方案基本功能的关键难点体现在三个方面：

1．PC 客户端持续发起域名访问服务。

PC 客户端访问网站时使用域名访问，需要进行 DNS 查询翻译成 IP 地址。由于实验需要模拟 PC 客户端对恶意网站黑名单“blacklist.txt”列表进行访问。因此，客户端将使用 Scapy[8]每隔 10 秒生成“blacklist.txt”列表中恶意网站域名访问包，来模拟 PC 客户端对黑名单内多个网站进行访问的情况。

核心代码如下：

```
#########################################################################
dnsv1.py    持续发起域名访问服务
#########################################################################
#!/usr/bin/python
from scapy.all import *
from time import sleep

fd = open("blacklist.txt",'r')
i = 1
dns_server = "222.200.160.1"
while True:
    q = fd.readline().strip()    # use strip to take off '\n'

    if not q: # to the end of file, url is null
        break
    print i ,q
    i += 1
    send(IP(dst=dns_server)/UDP(sport=RandShort())/DNS
    (qr=0,opcode=0,rd=1,qd=DNSQR(qname=q,qtype=1,qclass=1)))
    sleep(10)
```

2．SDN 控制器对所有交换机上送的 DNS 查询报文进行捕获和解析，对报文中请求解析的网址合法性进行判断。

(a)捕获：

OpenFlow Switch 会收到很多的数据包，对于没有流表可循的流会询问 SDN 控制器，因此，在我们的 firewallv2 模块中，我们在每个 switch 连接时就下发流表，让 DNS request 包直接送到控制器。接下来，对 OpenFlow Switch 送上来的 Packet_in 消息进行筛选，对于 DNS 请求的数据包，我们会调用 checkBlackList 函数，查询由恶意网站黑名单“blacklist.txt”列表生成的 Bloom Filter。

核心代码如下：

```
#########################################################################
firewallv2.py    控制器在各 Switch 连接建立时下发流表项，让 Switch 接收到的
                 DNS request 包直接送到控制器
#########################################################################
```

```
def _handle_ConnectionUp (event):
  global connectionUpNum
  if connectionUpNum == 0:    # initial the bloomfilter only once
    _init_bloomfilter()
  connectionUpNum += 1

  #sw = core.openflow.connections.keys()
  msg = of.ofp_flow_mod()
  #msg.priority = 65535
  msg.match = of.ofp_match()
  msg.match.dl_type = pkt.ethernet.IP_TYPE
  msg.match.nw_proto = pkt.ipv4.UDP_PROTOCOL
  msg.match.tp_dst = 53
  msg.actions.append(of.ofp_action_output(port = of.OFPP_CONTROLLER))
  event.connection.send(msg)

#######################################################################
firewallv2.py   DNS 请求的数据包捕获
#######################################################################
    p = event.parsed.find('dns')
     if p is not None and p.parsed and p.qr == 0: # qr=0 means DNS request
        eth_pkt = event.parsed
        if checkBlackList (p,eth_pkt.src,eth_pkt.payload.srcip):   # if it's URL in
BlackList, just drop it.
          return
```

(b)解析与合法性判断：

考虑到黑名单的大小数以万计，为了提高查询的效率以及减少内存的消耗，在我们的 firewallv2 模块中，我们使用了 Bloom Filter 这种空间效率很高的随机数据结构，把恶意网站黑名单“blacklist.txt”生成一个 Bloom Filter，伴随着模块的导入而初始化。

核心代码如下：

```
#######################################################################
BloomFilte.py   Bloom Filter 数据结构见附录 1
#######################################################################

#######################################################################
firewallv2.py   恶意网站黑名单“blacklist.txt” —BloomFilter 初始化
#######################################################################
def _init_bloomfilter():
  global bf
  fd = open("./ext/ blacklist.txt",'r')
  i = 0
  while True:
```

```
            url = fd.readline().strip()    # use strip to take off '\n'
            if not url: # to the end of file, url is null
            break
            if bf.has_element(url) == False:
                bf.insert_element(url)
            else:
                i = i+1
                print '%d url :%s has exist' % (i,url)
    print "BloomFilter is ready."
    fd.close()

#########################################################################
firewallv2.py    DNS 请求的合法性判断
#########################################################################
def checkBlackList (p,srcMAC,srcIP):
    global bf,cnt

    for q in p.questions:
        if q.qclass == 1 and ( q.qtype == 1 or q.type == 28):
# q.class=1, means q.class=IN. q.qtype=1, means q.qtype=A, is ipv4;
# q.qtype=28 or 0x1c means q.qtype=AAAA, is ipv6
                        #a = q.name.strip(".sysu.edu.cn")
            if bf.has_element(q.name) == True:
        cnt += 1
                f = open("./ext/firewall.log",'a')
                f.write(str(cnt)+" find URL in BlackList: "+q.name+'\n')
                f.write("from mac: "+str(srcMAC)+", ip:"+str(srcIP)+'\n')
                f.write("time: "+str(time.ctime())+'\n\n')
                f.close()
                print "%d find URL in BlackList: %s" % (cnt,q.name)
                print "from mac: %s, ip: %s" % (srcMAC,srcIP)
                print "time: %s\n" % time.ctime()
                return  True
            else: # not find
                return    False
    else: # if is not dns packet
            return False
```

3．SDN 控制器对 DNS 查询报文的合法性判断后的处理机制。看是否在黑名单中，如果在的话直接丢弃掉，如果不在的话，保证正常网站也可以访问。

动态流表下发：

这里我们改写了 l2_learning 这个模块。我们调用了 LearningSwitch 这个类，当我们看到一个包，我们想输出端口将最终导致目的地。为此，我们构建一个表映射地址到 OpenFlow Switch 端口。

对于每一个从交换机送上来的数据包，整个的处理流程如下：
(1)使用源地址和交换机端口更新表地址/端口。
(2)判断 transparent = False 和以太网类型是否 LLDP 或者包的目的地址是桥过滤地址；是的话，包丢弃。
(3)判断目的地址是组播？是的话，包泛洪。
(4)判断端口目的地在我们的地址/端口表里面？不是的话，包泛洪。
(5)判断输出端口和输入端口是否一样？是的话，包丢弃。
(6)把流表安装到各台 OpenFlow Switch 上，使得 OpenFlow 交换机适当的端口发送数据包。

核心代码如下：

```
#######################################################################
firewallv2.py    DNS 查询报文的合法性判断后的处理机制
#######################################################################
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
        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)
```

2.2 恶意网站防护方案扩展功能的实现（创新）

2.2.1 方案扩展功能系统框架及工作流程

（一）方案扩展功能系统框架

恶意网站防护方案扩展功能系统框架如图 2-3 所示，在原有基本功能的基础上，增加了白名单检测模块、名单修改模块以及可疑域名智能检测模块。

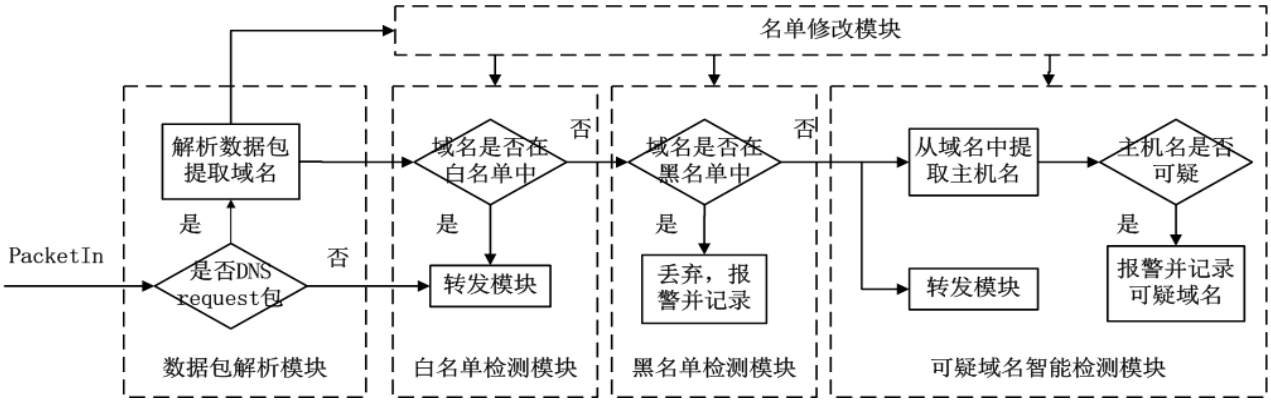

图 2-3　恶意网站防护方案扩展功能系统框架

1.白名单检测模块

将可信域名列表读入，逐条域名存放到 CBF(CountingBloomFilter)[9]中，形成 WL_CBF(WhiteList CBF)。对解析得到的域名进行多个哈希运算，判断其是否在 WL_CBF 中。若在，POX 则命令 switch 将 DNS 数据包发送出去；若不在，则将域名送入黑名单检测模块。

数据来源：

GFW 白名单 https://github.com/n0wa11/gfw_whitelist/blob/master/whitelist.pac

我们以这份白名单为基础，用网络爬虫获取了每个可信网址首页上的连接，经过去重复、删除恶意网址等处理后，获得的白名单包含 21371 个域名。

2.名单修改模块

根据 opcode 字段[10]，将域名的信息添加到相应的 CBF 中，或者从相应的 CBF 删除域名的信息。opcode 的数值与对应的操作如下表 2-1 所示：

表 2-1　opcode 的数值与对应的操作

opcode 数值	操作
10	将域名添加到 BL_CBF 中
11	从 BL_CBF 中删除域名
12	将域名添加到 WL_CBF 中
13	从 WL_CBF 中删除域名
14	将域名2添加到 TLD_CBF 中
15	从 TLD_CBF 中删除域名3

[2] 这里其实是 TLD 或 SLD，而不是完整的域名。

在将域名添加到 CBF 中之前，会先判断域名是否在 CBF 中，若在，则不添加；若不在，则添加。在从 CBF 中删除域名之前，会先判断域名是否在 CBF 中，若在，则删除；若不在，则不执行删除操作。

3.可疑域名智能检测模块

(a) 域名组成

一个完整的域名由二个或二个以上部分组成，各部分之间用英文 的句号"."来分隔，例如：

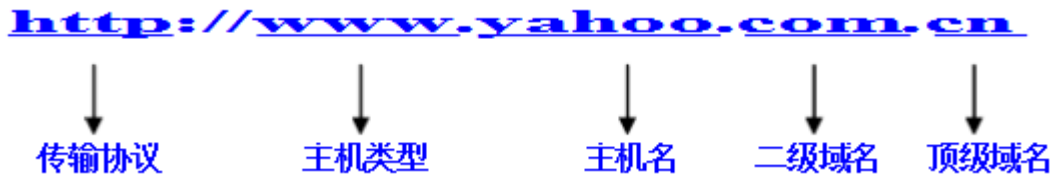

图 2-4 完整域名例子

在一个完整的域名中，最后一个"."的右边部分称为顶级域名（Top Level Domain, TLD），在上面的域名例子中，com、us 和 uk 是顶级域名。最后一个"."的左边部分称为二级域名（SLD），例如，域名 yahoo.com 中 yahoo 是二级域名，域名 yahoo.ca.us 中 ca 是二级域名，而域名 yahoo.co.uk 中 co 是二级域名。二级域名的左边部分称为三级域名，三级域名的左边部分称为四级域名，以此类推。

低一级的所有域名都只能由高一级的域名来分配，如 music.baidu.com、map.baidu.com 等只能由 baidu.com 来分配。因而，我们认为，只要去掉主机名是可信，其对应的下一或多级域名也是可信；如果主机名是恶意或可疑的，则其下一或多级域名也是可疑的，baidv 可疑，则认为 map.baidv.com 是可疑的。

(b) 主机名提取

因此，对于每一个待检测的域名，为了得到其主机名，我们必须去掉其后面的 TLD 和 SLD(如果有 SLD)。

TLD 包括通用 TLD（com、org、net 等），国家 TLD（cn、us、uk 等）；SLD 包括 TLD 下定义的通用 SLD（如 com、edu、co 等），国内省份（bj、gd 等）。我们使用的 TLD.txt 包括 TLD 和 SLD 共 304 个。我们为 TLD/SLD 建立一个 Counting Bloom Filter，这个相比黑白名单，规模小很多。

我们会‘.’为分界符，将域名分成多个字符串，并从最后一个字符串开始，查看当前字符串是否在 TLD/SLD 的 Counting Bloom Filter 中。如果在，则继续查询下一个字符；如果不在，由于是从右往左查询，可以断定当前字符串就是主机名。接着将主机名送到另一个函数，进行可疑程度的检测。

(c) 可疑域名检测

很多恶意或者可疑域名，如 www.lcbc.com、www.1cbc.com，与真实的可信域名 www.icbc.com，只是在主机名部分有不起眼的差异，即 lcbc、1cbc 和 icbc 很相似。

这些主机名间的特点是编辑距离[11, 12]很小，但不为零。我们从白名单中提取所有的主机名，存放 BK-tree[13]这种数据结构中，BK-tree 在每次访问 5%~8%

[3]这里其实是 TLD 或 SLD，而不是完整的域名。

个节点[4]的情况下就能找到编辑距离最近的节点，并得到最小的编辑距离minDist，计算开销非常小，对于编辑距离，我们采用动态规划算法来求解，也能有效降低计算开销。当$\alpha \leq \min Dist \leq \beta$时，认为域名是可疑的；否则认为域名是可信的。

（二）方案扩展功能工作流程

恶意网站防护方案扩展功能工作流程如基本功能流程基本一致：

1．终端发起 DNS 请求

2．OpenFlow Switch 收到 DNS 请求，发起带有查询信息的报文封装成 Packet_in 的消息给 SDN 控制器。

3．SDN 控制器根据 OpenFlow Switch 上发的报文进行包解析，先与本地的白名单进行匹配，匹配成功则进行动态流表分发，匹配失败则与本地的黑名单进行匹配。匹配成功，则对黑名单内的域名的包进行丢弃，匹配失败则进入可疑域名智能检测系统。首先，进入可疑域名智能检测系统的域名去 TLD\SLD，然后对 hostname（主机名）进行相似度对比，如果其编辑距离在 1 到 2 之间的话，则报警并记录可疑域名，并且进行动态流表分发，使之接入校园网，经过网关接入 Internet，实现 DNS 请求响应。

2.2.3 方案扩展功能关键难点

在恶意网站防护方案基本功能的基础上，扩展功能对解析与合法性判断提出更高的要求，关键难点体现在三个方面：

1．白名单检测模块的引入

核心代码如下：

```
#########################################################################
CountingBloomFilte.py    CountingBloom Filter 数据结构见附录 2
#########################################################################

#########################################################################
firewallv8.py    恶意网站白名单“whitelist.txt” —Counting Bloom Filter 初始化
#########################################################################
## White List CBF
   f = open("./ext/whitelist.txt",'r') # the path has to be changed!!
   i = 0
   while True:
            url = f.readline().strip()    # use strip to take off '\n'
     if not url: # to the end of file, url is null
           break
```

[4] 若用列表存放所有可信主机名，对每个待检测的主机名，需要遍历列表中所有节点，计算待检测主机名与所有可信主机名的编辑距离，才能得到 minDist，可算开销非常大。

```
            if wl_cbf.has_element(url) == False:
                wl_cbf.insert_element(url)
            #else:
      #      i = i+1
               #    print '%d url :%s has exist in Whitelist' % (i,url)
   print "CountingBloomFilter for Whitelist is ready."
   print time.ctime()
   f.close()
```

2．名单修改模块

针对不同的业务要求，我们可以对黑名单、白名单、TLD 名单域名进行增加或者减少的功能。以白名单为例，白名单域名的增加与减少的核心代码分别如下：

```
######################################################################
firewallv8.py   网站白名单“whitelist.txt” 域名添加与减少核心代码
######################################################################

def addURLtoWLCBF(p,srcMAC,srcIP):
   global add_wl_cnt, wl_cbf
   for q in p.questions:
     add_wl_cnt += 1
     if wl_cbf.has_element(q.name) == False:
                wl_cbf.insert_element(q.name)
          print "\nadd URL in WhiteList: %s" % q.name
          print "from mac: %s, ip: %s" % (srcMAC,srcIP)
          print "time: %s" % time.ctime()
          f = open("./ext/addWLURL.log",'a')
          f.write(str(add_wl_cnt)+" add URL in WhiteList: "+q.name+'\n')
          f.write("from mac: "+str(srcMAC)+", ip:"+str(srcIP)+'\n')
          f.write("time: "+str(time.ctime())+'\n\n')
          f.close()
     else:
          print "Add error! %s has already been in WhiteList." % q.name

def deleteURLinWLCBF(p,srcMAC,srcIP):
   global del_wl_cnt, wl_cbf
   for q in p.questions:
     del_wl_cnt += 1
     if wl_cbf.has_element(q.name) == True:
                wl_cbf.delete_element(q.name)
          print "\ndelete URL in WhiteList: %s" % q.name
          print "from mac: %s, ip: %s" % (srcMAC,srcIP)
          print "time: %s" % time.ctime()
          f = open("./ext/deleteWLURL.log",'a')
```

```
            f.write(str(del_wl_cnt)+" delete URL in WhiteList: "+q.name+'\n')
            f.write("from mac: "+str(srcMAC)+", ip:"+str(srcIP)+'\n')
            f.write("time: "+str(time.ctime())+'\n\n')
            f.close()
        else:
            print "Delete error! %s is not in WhiteList." % q.name
```

3．可疑域名的解析与合法性判断

可疑域名的解析与合法性判断分两步。

(a)先经过白名单和黑名单的过滤，如果域名在白名单内，直接转发，如果域名不在白名单，则进入黑名单的过滤，如果域名在黑名单内，则直接丢弃，如果域名不在黑名单，则进入可疑域名智能检测系统。

核心代码如下：

```
#########################################################################
firewallv8.py   黑白名单过滤 “checkURL”功能函数
#########################################################################
def checkURL (p,srcMAC,srcIP):
  global bl_cbf,bl_cnt,wl_cbf,wl_cnt

  for q in p.questions:
    if q.qclass == 1 and ( q.qtype == 1 or q.type == 28):
      if wl_cbf.has_element(q.name) == True: # Is url in whitelist?
  #wl_cnt += 1
        #print "%d %s is in wl" % (wl_cnt,q.name)
        return False
      elif bl_cbf.has_element(q.name) == True: # Is url in blacklist?
        bl_cnt += 1
        f = open("./ext/firewall.log",'a')
        f.write(str(bl_cnt)+" find URL in BlackList: "+q.name+'\n')
        f.write("from mac: "+str(srcMAC)+", ip:"+str(srcIP)+'\n')
        f.write("time: "+str(time.ctime())+'\n\n')
        f.close()
        print "\n%d find URL in BlackList: %s" % (bl_cnt,q.name)
        print "from mac: %s, ip: %s" % (srcMAC,srcIP)
        print "time: %s" % time.ctime()
        return  False
      else: # not find
        #return   False
        return checkhostname(q.name,srcMAC,srcIP)
    else: # if is not dns packet
      return False
```

(b)进入可疑域名智能检测系统的域名首先去 TLD\SLD，然后对 hostname 进

行相似度对比，如果其编辑距离在 1 到 2 之间的话，则报警并记录可疑域名。

```
#########################################################################
BKtree.py   BK-Tree 数据结构见附录 3
#########################################################################

#########################################################################
firewallv8.py  域名去 TLD\SLD 以及 hostname 相似度比较， “checkhostname” 以及“checkinBKtree” 功能函数
#########################################################################
def checkhostname(url,srcMAC,srcIP):
   #print "checkin hostname %s" % url
   hostname = url.split('.')
   n = len(hostname)
   for i in range(1,n+1):    # [1,2,3,...,n]
      if tld_cbf.has_element(hostname[n-i]):
         continue
      else:
         if checkinBKtree(hostname[n-i]):
            global tree_cnt
            tree_cnt += 1
            f = open("./ext/suspectURL.log",'a')
            f.write(str(tree_cnt)+" suspect: "+url+'\n')
            f.write("from mac: "+str(srcMAC)+", ip:"+str(srcIP)+'\n')
            f.write("time: "+str(time.ctime())+'\n\n')
            f.close()
         return False

def checkinBKtree(name):
   #print "checkin BKtree %s " % name
   global name_tree, afa, beta      # afa = 1, beta = 2
   [minDist,minData] = name_tree.findMinDist(name,beta)
   if minDist >= afa and minDist <= beta:
      print "suspect url: %s, similar to: %s, minEditDist: %d" % (name, minData, minDist)
      return True # suspect
   else:
      #print "normal url: %s, minData: %s, minDist: %d" % (name, minData, minDist)
      return False # not suspect
```

2.3 基本功能与扩展功能的数据结构及算法说明

2.3.1 黑白名单域名存储数据结构

黑白名单的对比需要判断域名是否在黑白名单中，域名是字符串，名单是字符串集合，如果用字符串的对比方法，将域名与名单中的域名逐个比对，时间开销非常大。另外，黑白名单通常包含上万个域名，规模大的甚至包含上百万个域名。如果直接用字符串集合的方式来保存名单，存储空间的开销非常大。

基于以上两点考虑，我们采用 Counting Bloom Filter 来存储黑白名单，能有效减少存储空间开销，同时将对比的时间开销减低为常数。Counting Bloom Filter 由 Bloom Filter 演变而来，我们先介绍 Blomm Filter，再介绍 Counting Bloom Filter。

1．Bloom Filter

Bloom Filter 是一种空间效率很高的随机数据结构，它利用位数组很简洁地表示一个集合，并能判断一个元素是否属于这个集合。Bloom Filter 的这种高效是有一定代价的：在判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误认为属于这个集合（false positive）。因此，Bloom Filter 不适合那些“零错误”的应用场合。而在能容忍低错误率的应用场合下，Bloom Filter 通过极少的错误换取了存储空间的极大节省。但 Bloom Filter 没有 false negative，不会把属于集合中的元素误认为不属于这个集合。

Bloom Filter 用位数组表示集合的[5]。初始状态时，Bloom Filter 是一个包含 m 位的位数组，每一位都置为 0。

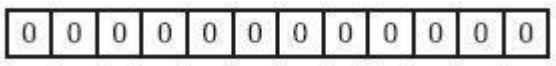

图 2-5 Bloom Filter 初始状态

为了表达 $S=\{x_1,x_2,\ldots,x_n\}$这样一个 n 个元素的集合，Bloom Filter 使用 k 个相互独立的哈希函数（Hash Function），它们分别将集合中的每个元素映射到 $\{1,\ldots,m\}$的范围中。对任意一个元素 x，第 i 个哈希函数映射的位置 hi(x)就会被置为 1（1≤i≤k）。注意，如果一个位置多次被置为 1，那么只有第一次会起作用，后面几次将没有任何效果。在下图中，k=3，且有两个哈希函数选中同一个位置（从左边数第五位）。

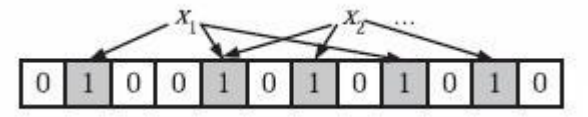

图 2-6 x1 和 x2 存入 Bloom Filter

[5] 由于控制器是 POX，我们用 python 的 BitVector，即位数组结构来实现 Bloom Filter，BitVector 中每一位就是 1bit。

在判断 y 是否属于这个集合时，对 y 应用 k 次哈希函数，如果所有 hi(y)的位置都是 1（1≤i≤k），那就认为 y 是集合中的元素，否则就认为 y 不是集合中的元素。下图中 y1 就不是集合中的元素。y2 或者属于这个集合，或者刚好是一个 false positive。故无论 Bloom Filter 中存放多少个元素，当要查询某个元素是否在 Bloom Filter 中时，只需进行 k 次哈希运算。

图 2-7 Bloom Filter 查询示例

在建立一个 Bloom Filter，有下列参数需要考虑：

假设集合元素个数为 n，要求的 false positive 不超过 ε，则所需的 Bloom Filter 的位数组位数为至少为：

$$m \geq n \frac{\log_2(1/\varepsilon)}{\ln 2} = n \log_2 e \cdot \log_2(1/\varepsilon) \tag{1}$$

所需的哈希函数个数为：

$$k = \ln 2 \cdot (m/n) \tag{2}$$

所采用的哈希函数为 BKDRHash，依据[14]，BKDRHash 是常见的处理字符串的哈希函数中综合评分最高的。

我们所用的 Bloom Filter 对外提供的函数主要包含：新建一个 Bloom Filter、插入一个元素 insert_element()、查询是否包含某元素 has_element()。

2．Counting Bloom Filter

有时需要对黑白名单中的域名进行增删，但 Bloom Filter 不支持删除操作，于是我们用 Counting Bloom Filter 代替 Bloom Filter。假设 Bloom Filter 有 m bit，将每一 bit 看做一个计数器，则 Bloom Filter 的每个计数器的取值为 0 或 1，而对应的 Counting Bloom Filter 的每一个计数器的取值为 $0 \sim l$，则每个计数器需要 $t = \log_2 l$ bits。在插入元素时给对应的 k 个计数器的值分别加 1，删除元素时给对应的 k 个计数器的值分别减 1。

[9]表明，$t = 4$ 对于大多数应用程序来说已经足够，此时任意计数器的值超过 16（即溢出）的概率为 $p \leq 1.37 \times 10^{-15} \times m$。我们在自己实现的 Counting Bloom Filter 中，也令 $t = 4$，即位数组为 Bloom Filter 的 4 倍。

建立一个 Counting Bloom Filter 时需要考虑的参数如下：

假设集合元素个数为 n，要求的 false positive 不超过 ε，则所需的 Bloom Filter 的位数组位数为至少为：

$$m \geq 4 \cdot n \frac{\log_2(1/\varepsilon)}{\ln 2} = 4 \cdot n \log_2 e \cdot \log_2(1/\varepsilon) \tag{3}$$

所需的哈希函数个数为：

$$k = \ln 2 \cdot (m/n) = 4 \cdot \ln 2 \cdot \log_2 e \cdot \log_2(1/\varepsilon) \tag{4}$$

我们实现的 Counting Bloom Filter 对外提供的函数主要包含：新建一个 Counting Bloom Filter、插入一个元素 insert_element()、查询是否包含某元素 has_element()、删除一个元素 delete_element()。

Counting Bloom Filter 所需的位数组数 m 和哈希函数个数 k 随着元素个数 n 和虚警率 ε 的变化如图 2-8、2-9 所示。位数组中每一位占用 1 bit 的内存，当需要存储 100 万个域名，虚警率为 0.0001 时，即大规模域名数量又低 false positive rate 的情况下，仅需要 6.34MB 的存储空间和 37 个哈希函数。实验中，我们发现，在生成 Counting Bloom Filter，只要设置的 n 值比实际需要存储的域名数很多，则总是 ε=0.1，也基本不会出现 false positive。所以，在扩展功能版中，我们采用的黑白名单 Counting Bloom Filter，所设置的参数为 n=10 万，ε=0.1，对应需要 158KB 存储空间和 10 个哈希函数。如果考虑到系统的大规模应用，则需要将 n、ε 的值作相应变化。

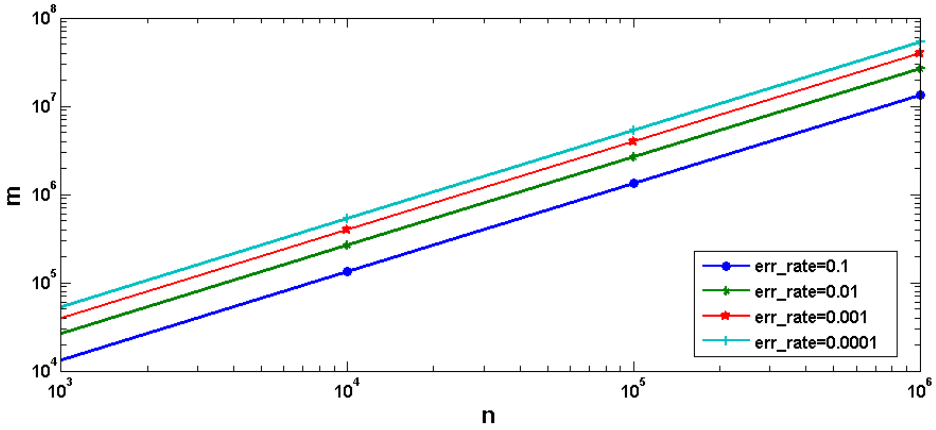

图 2-8 m 随 n、ε 的变化趋势

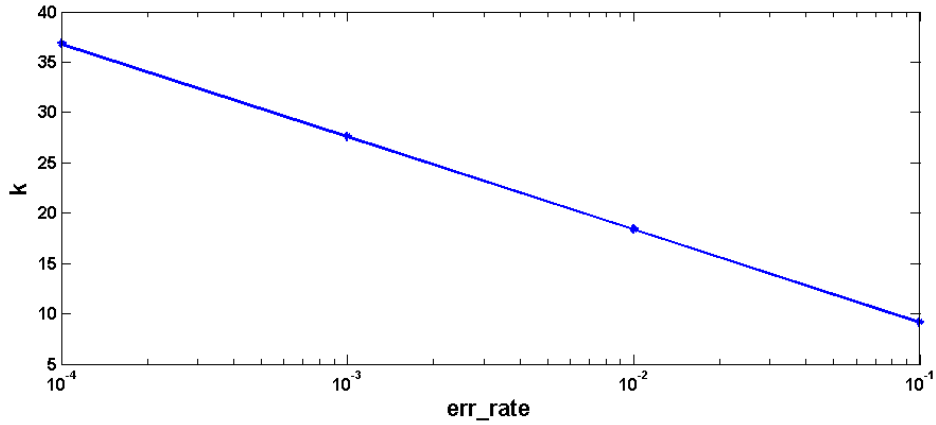

图 2-9 k 随 ε 的变化趋势

2.3.2 可疑域名检测系统数据结构

1．Counting Bloom Filter

与白黑名单模块的相同，只是放到其中的是通用的 TLD/SLD，用于对每一个待检测的域名，进行主机名提取，去掉其后面的 TLD 和 SLD(如果有 SLD)。n=1000，ε=0.1，对应需要 1.62KB 存储空间和 10 个哈希函数。

2．BK-Tree[15]

1965 年，俄国科学家 Vladimir Levenshtein 给字符串相似度做出了一个明确的定义叫做 Levenshtein 距离，我们通常叫它“编辑距离”。字符串 A 到 B 的编辑距离是指，只用插入、删除和替换三种操作，最少需要多少步可以把 A 变成 B。例如，从 FAME 到 GATE 需要两步（两次替换），从 GAME 到 ACM 则需要三步（删除 G 和 E 再添加 C）。

如图所示，首先我们随便找一个单词作为根（比如 GAME）。以后插入一个单词时首先计算单词与根的 Levenshtein 距离：如果这个距离值是该节点处头一次出现，建立一个新的儿子节点；否则沿着对应的边递归下去。例如，我们插入单词 FAME，它与 GAME 的距离为 1，于是新建一个儿子，连一条标号为 1 的边；下一次插入 GAIN，算得它与 GAME 的距离为 2，于是放在编号为 2 的边下。再下次我们插入 GATE，它与 GAME 距离为 1，于是沿着那条编号为 1 的边下去，递归地插入到 FAME 所在子树；GATE 与 FAME 的距离为 2，于是把 GATE 放在 FAME 节点下，边的编号为 2。

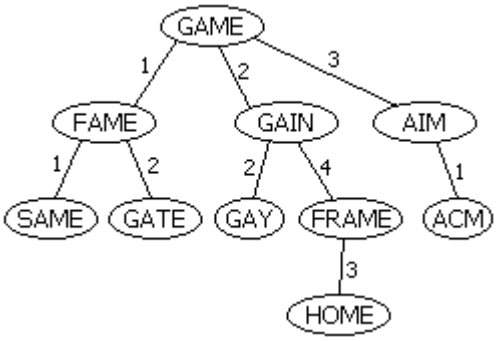

图 2-10　BK-tree 示例

查询操作异常方便。按照建树的原理，待查询的单词与当前节点比较后，得到距离 d，接着进入与当前节点距离同样为 d 的子节点，继续对比查询，直到发现 d=0 的节点（字符串相同）或到达叶子节点。由于每次与某个节点进行比较时都可以排除很多子树，每次查询的开销较低。

查询操作异常方便。如果我们需要返回与错误单词距离不超过 n 的单词，这个错误单词与树根所对应的单词距离为 d，那么接下来我们只需要递归地考虑编号在 d-n 到 d+n 范围内的边所连接的子树。由于 n 通常很小，因此每次与某个节点进行比较时都可以排除很多子树。

实践表明，一次查询所遍历的节点不会超过所有节点的 5%到 8%，效率远远超过暴力枚举。如 TLD.txt 文档中有 5336 个主机名，待查询的域名为 baidv 时，只需与 385 个节点进行比较，即得出 baidv 与 baidu 的编辑距离最小（为 1），访问节点数比例为 7.2%。

第三章 实验过程及结果

3.1 恶意网站防护方案基本功能的实现

3.1.1 OpenFlow 实验平台初始化

1.PC-A 初始化

#IP 配置(ifconfig)
ifconfig eth0 192.168.100.11

#初始化版卡
/usr/local/sbin/cpci_reprogram.pl –all

#下载 OpenFlow bit 文件
nf_download /usr/local/netfpga/bitfiles/openflow_switch.bit

#设置 openflow datapath(路径 /usr/local/netfpga/openflow/udatapath)
cd /usr/local/netfpga/openflow/udatapath
./ofdatapath --detach punix:/var/run/dp0 -d 40169FF344C0 -i nf2c0,nf2c1,nf2c2,nf2c3

#与远程 Controller 进行通信(路径/usr/local/netfpga/openflow/secchan)
cd /usr/local/netfpga/openflow/secchan
./ofprotocol unix:/var/run/dp0 tcp:192.168.100.100:6633

#显示流表信息
cd /usr/local/netfpga/openflow/utilities
./dpctl dump-flows unix:/var/run/dp0

2.PC-B 初始化

#IP 配置(ifconfig)
ifconfig eth0 192.168.100.22

#初始化版卡
/usr/local/sbin/cpci_reprogram.pl –all

#下载 OpenFlow bit 文件
nf_download /usr/local/netfpga/bitfiles/openflow_switch.bit

#设置 openflow datapath(路径 /usr/local/netfpga/openflow/udatapath)
cd /usr/local/netfpga/openflow/udatapath

./ofdatapath --detach punix:/var/run/dp0 -d 40169FF344C0 -i nf2c0,nf2c1,nf2c2,nf2c3

#与远程 Controller 进行通信(路径/usr/local/netfpga/openflow/secchan)
cd /usr/local/netfpga/openflow/secchan
./ofprotocol unix:/var/run/dp0 tcp:192.168.100.100:6633

#显示流表信息
cd /usr/local/netfpga/openflow/utilities
./dpctl dump-flows unix:/var/run/dp0

3.PC-C 初始化
#IP 配置(ifconfig)
ifconfig eth0 192.168.100.44

#初始化版卡
/usr/local/sbin/cpci_reprogram.pl –all

#下载 OpenFlow bit 文件
nf_download /usr/local/netfpga/bitfiles/openflow_switch.bit

#设置 openflow datapath(路径 /usr/local/netfpga/openflow/udatapath)
cd /usr/local/netfpga/openflow/udatapath
./ofdatapath --detach punix:/var/run/dp0 -d 40169FF344C0 -i nf2c0,nf2c1,nf2c2,nf2c3

#与远程 Controller 进行通信(路径/usr/local/netfpga/openflow/secchan)
cd /usr/local/netfpga/openflow/secchan
./ofprotocol unix:/var/run/dp0 tcp:192.168.100.100:6633

#显示流表信息
cd /usr/local/netfpga/openflow/utilities
./dpctl dump-flows unix:/var/run/dp0

4. PC-0 初始化
#终端网络配置（gedit /etc/sysconfig/network-scripts/ifcfg-eth1）
# Realtek Semiconductor Co., Ltd. RTL-8169 Gigabit Ethernet
DEVICE=eth1
BOOTPROTO=none
TYPE=Ethernet
NM_CONTROLLED=yes
ONBOOT=yes
USERCTL=no
IPV6INIT=no
NM_CONTROLLED=yes
DEFROUTE=yes

IPV4_FAILURE_FATAL=yes
NAME="System eth1"
UUID=9c92fad9-6ecb-3e6c-eb4d-8a47c6f50c04
IPADDR=172.18.216.49　#校园网 IP 地址
NETMASK=255.255.255.0
GATEWAY=172.18.216.254　#校园网网关地址
PREFIX=24
DNS1=222.200.160.1　　#校园网 DNS 地址

5.PC-1 初始化

#终端网络配置（gedit /etc/sysconfig/network-scripts/ifcfg-eth1）
# Please read /usr/share/doc/initscripts-*/sysconfig.txt
# for the documentation of these parameters.
DEVICE=eth1
BOOTPROTO=none
TYPE=Ethernet
HWADDR=68:05:CA:08:1F:C4
ONBOOT=yes
USERCTL=no
IPV6INIT=no
NM_CONTROLLED=no
PREFIX=24
DEFROUTE=yes
IPV4_FAILURE_FATAL=yes
NAME="System eth1"
UUID=9c92fad9-6ecb-3e6c-eb4d-8a47c6f50c04
IPADDR=172.18.216.42
NETMASK=255.255.255.0　#校园网 IP 地址
GATEWAY=172.18.216.254　#校园网网关地址
DNS1=222.200.160.1　　#校园网 DNS 地址

6.PC-D 初始化

#IP 配置(ifconfig)
ifconfig eth1 192.168.100.100

POX 下发流表环境配置，加载相关模块
./pox.py log --no-default samples.pretty_log web messenger \
messenger.log_service messenger.ajax_transport openflow.of_service \
poxdesk openflow.discovery poxdesk.tinytopo \
openflow.of_01 --address=192.168.100.100 --port=6633 \
firewallv2 py

如图 3-1 所示：

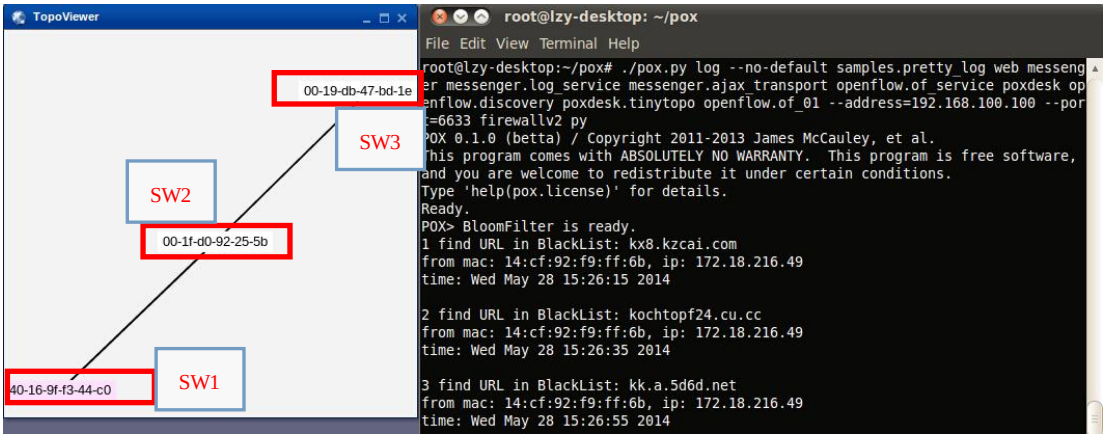

图 3-1 Switches 连接上 POX

SDN 控制器加载 firewallv2 模块，由恶意网站黑名单“blacklist.txt”列表生成的 Bloom Filter 已生成就绪。

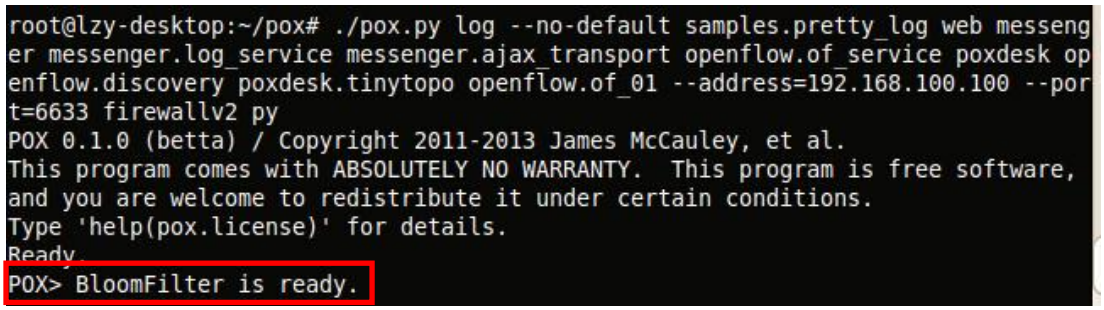

图 3-2 Bloom Filter 已生成就绪

3.1.2 实验结果及其现象（一）——恶意网站访问

客户端 PC-0，加载 dnsv1.py 模块，模拟 PC 客户端对恶意网站进行 DNS 访问请求，通过 Wireshark 观察 Eth1 端口数据包情况，如图 3-3 所示：

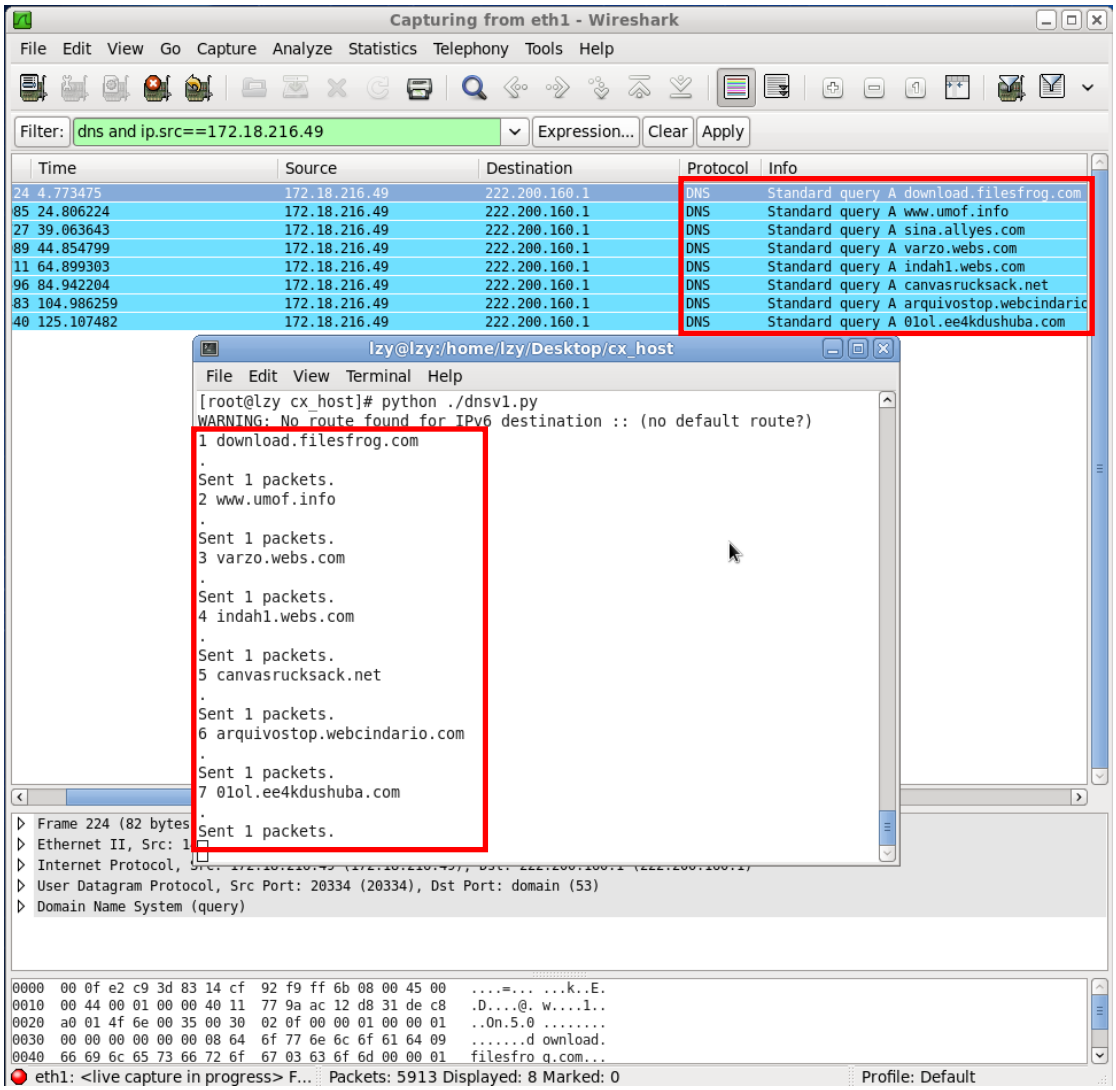

图 3-3 host 通过 Scapy 发送包含恶意域名的 DNS request

SDN 控制器检测到 OpenFlow 交换机送上来的 DNS 数据包，对数据包的解析可以发现是“download.filesfrog.com”，“www.umof.info”，“varzo.webs.com”，“indah1.webs.com”,“canvasrucksack.net”等网站的 DNS 访问请求，通过查询由黑名单生成的 BloomFilter，判断其为恶意网站，拒绝该网站的 DNS 解析请求服务。

SDN 控制器恶意网站访问警报信息如图 3-4 所示：

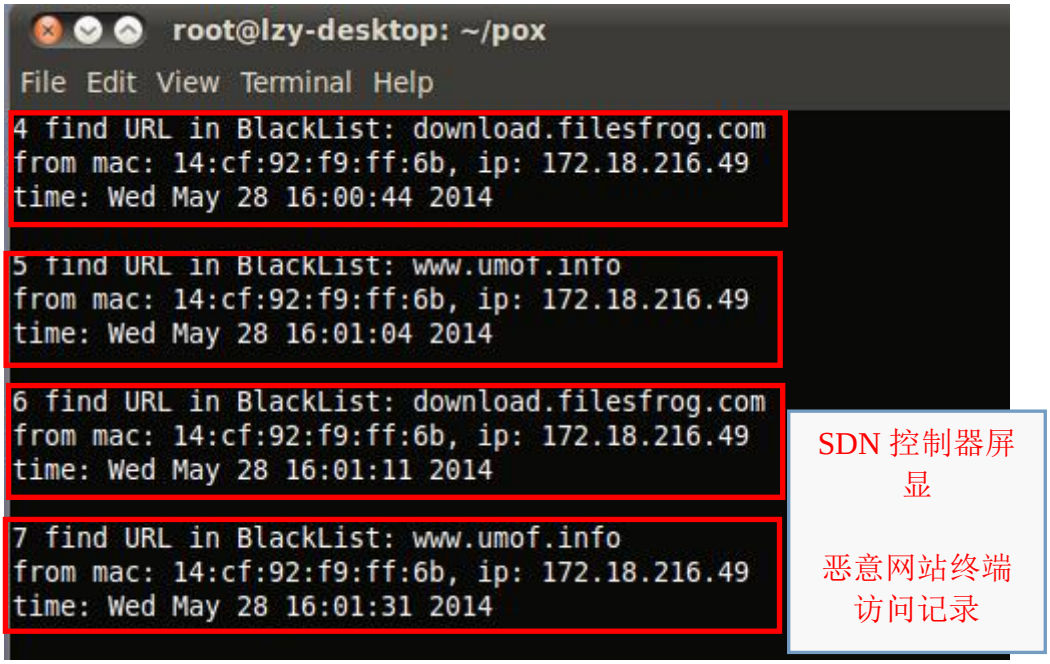

图 3-4 SDN 控制器上的恶意网站访问警报信息

3.1.3 实验结果及其现象（二）——正常网站访问

SDN 控制器加载 firewallv2 模块，由恶意网站黑名单“blacklist.txt”列表生成的 Bloom Filter 已生成就绪。

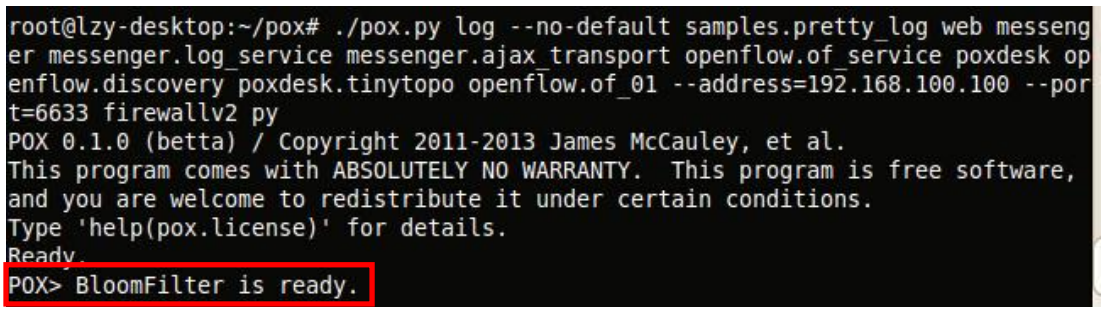

图 3-5 blacklist 的 Bloom Filter 就绪

客户端 PC-0 进行正常的网页访问，对“www.sina.com.cn”发出 DNS 访问请求，网页正常访问，通过 Wireshark 观察 Eth1 端口数据包情况，如图 3-6 所示：

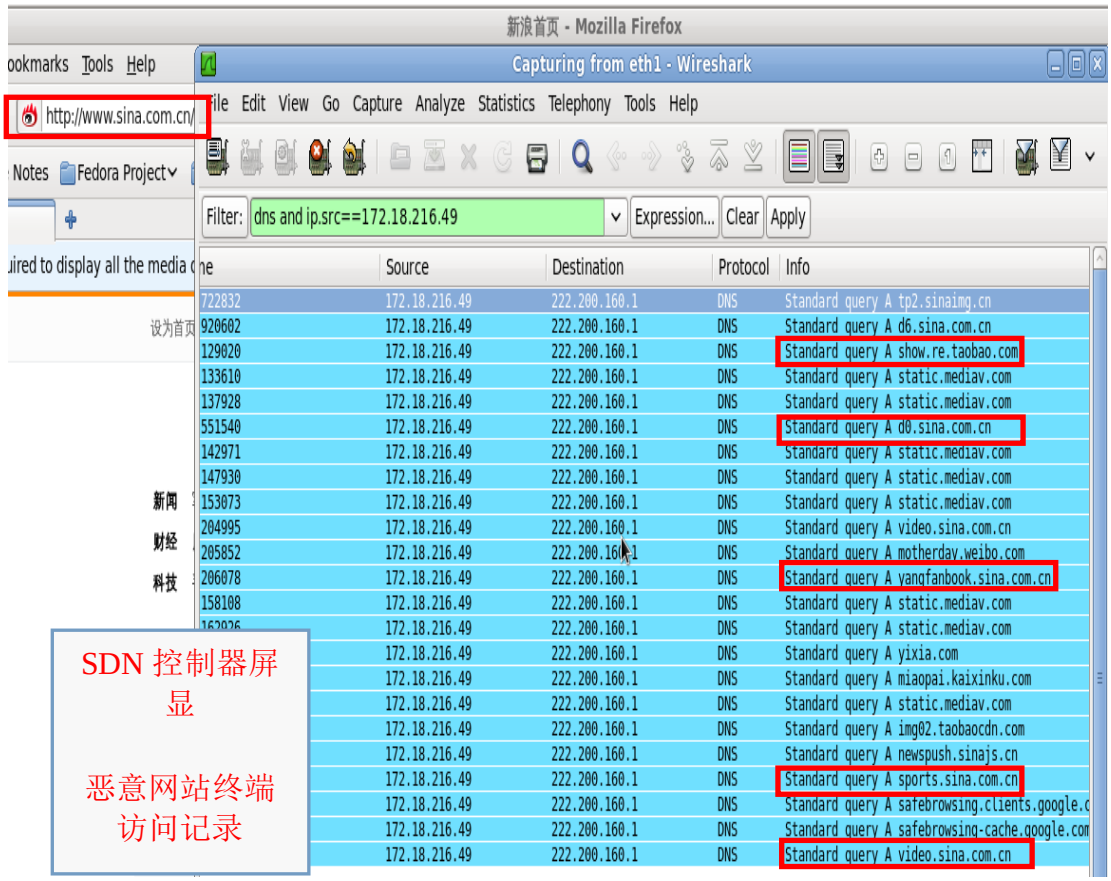

图 3-6 Wireshark 观察 Eth1 端口数据包情况

SDN 控制器检测到 OpenFlow 交换机送上来的 DNS 数据包，对数据包进行解析，可以发现 www.sina.com.cn 网页可以正常访问，基于 sina 的 DNS 服务请求经过查询黑名单生成的 BloomFilter，判断其不在恶意网站列表，因此允许该网站的 DNS 解析请求服务，并动态生成了流表项发给了 OpenFlow 交换机。

PC-A 上 Switch1 上流表如图 3-7 所示：

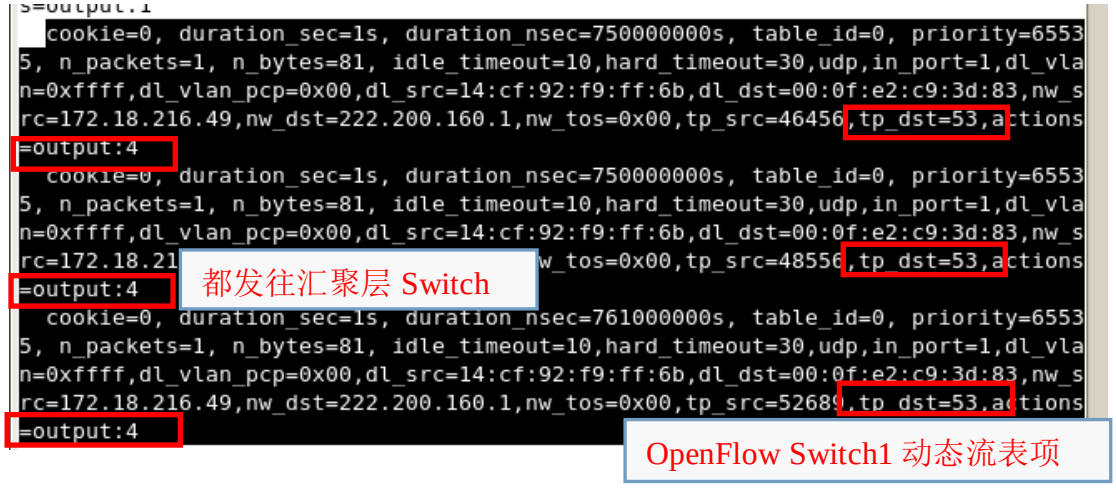

图 3-7 Switch 1 上的流表

PC-B 上 Switch2 上流表如图 3-8 所示：

图 3-8 Switch 2 上的流表

SDN 控制器恶意网站访问警报信息如图 3-9 所示：

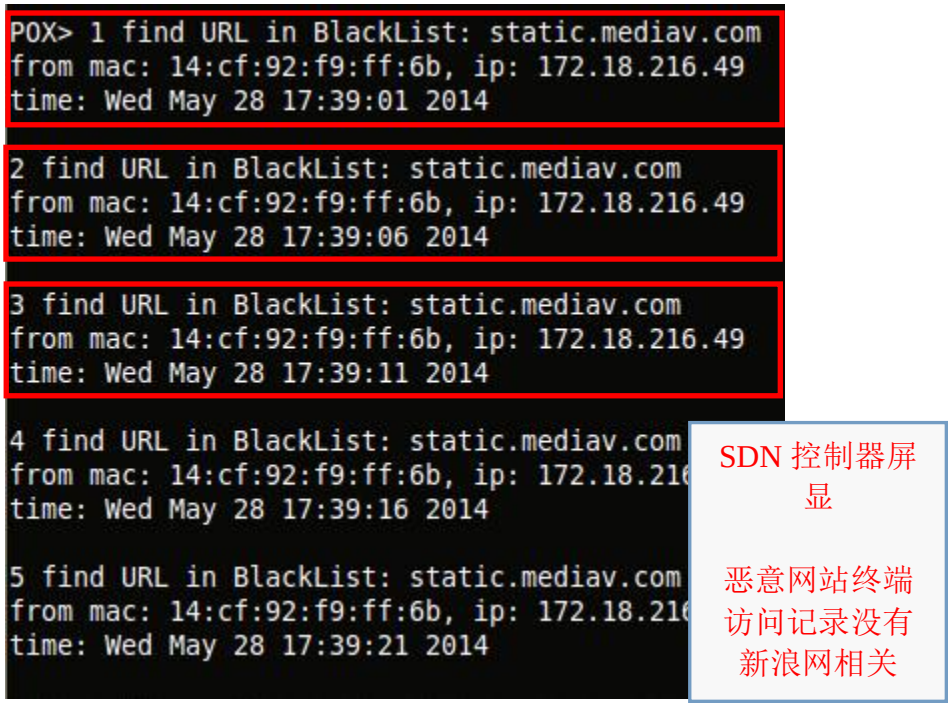

图 3-9　SDN 控制器恶意网站访问警报信息

3.2 恶意网站防护方案扩展功能的实现

3.2.1 OpenFlow 实验平台初始化

1.PC-A 初始化与基本功能的初始化情况一样
2.PC-B 初始化与基本功能的初始化情况一样
3.PC-C 初始化与基本功能的初始化情况一样
4. PC-0 初始化与基本功能的初始化情况一样
5.PC-1 初始化与基本功能的初始化情况一样
6.PC-D 初始化
#IP 配置(ifconfig)
ifconfig eth1 192.168.100.100

POX 下发流表环境配置，加载相关模块
./pox.py log --no-default samples.pretty_log web messenger \
messenger.log_service messenger.ajax_transport openflow.of_service \
poxdesk openflow.discovery poxdesk.tinytopo \
openflow.of_01 --address=192.168.100.100 --port=6633 \
firewallv8 py
如图 3-10 所示：

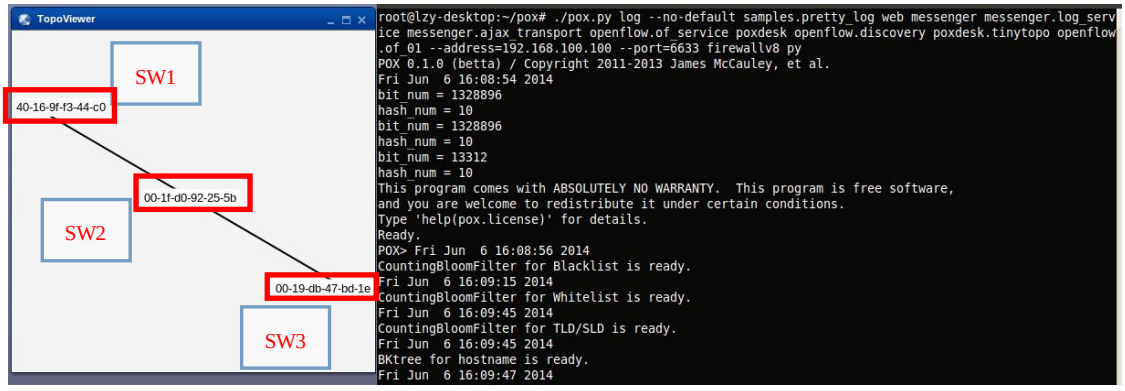

图 3-10 switches 连接上控制器

SDN 控制器加载 firewallv8 模块，由恶意网站黑名单“blacklist.txt”列表生成的 CountingBloomFilter 已生成就绪；由域名白名单“whitelist.txt”列表生成的 CountingBloomFilter 已生成就绪；由域名 TLD\SLD 名单“TLD.txt”列表生成的 CountingBloomFilter 已生成就绪；由域名主机名名单“hostname.txt”列表生成的 BKTree 已生成就绪。

图 3-11 黑名单、白名单、TLD\SLD 以及主机名名单初始化完毕

3.2.2 实验结果及其现象（一）——白名单网站访问

客户端 PC-0 进行正常的网页访问，对“www.sina.com.cn”发出 DNS 访问请求，网页正常访问，通过 Wireshark 观察 Eth1 端口数据包情况，如图 3-12 所示：

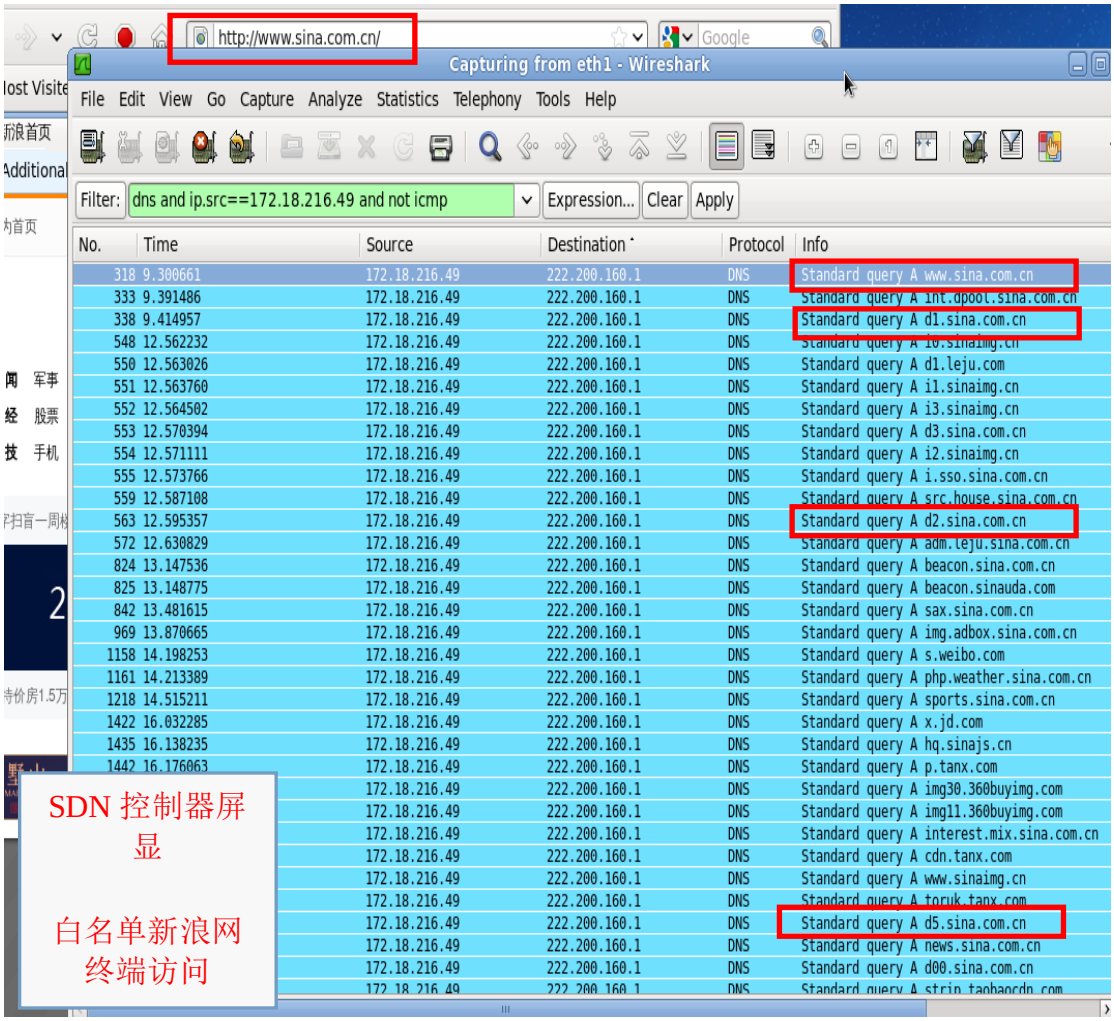

图 3-12 Wireshark 观察 Eth1 端口数据包情况

SDN 控制器检测到 OpenFlow 交换机送上来的 DNS 数据包，对数据包进行解析，可以发现是 www.sina.com.cn 网站的 DNS 访问请求，查询白名单，匹配成功，直接进入转发模块，网页可以正常访问，因此允许该网站的 DNS 解析请求服务，并动态生成了流表项发给了 OpenFlow 交换机。

PC-A 上 Switch1 上流表如图 3-13 所示：

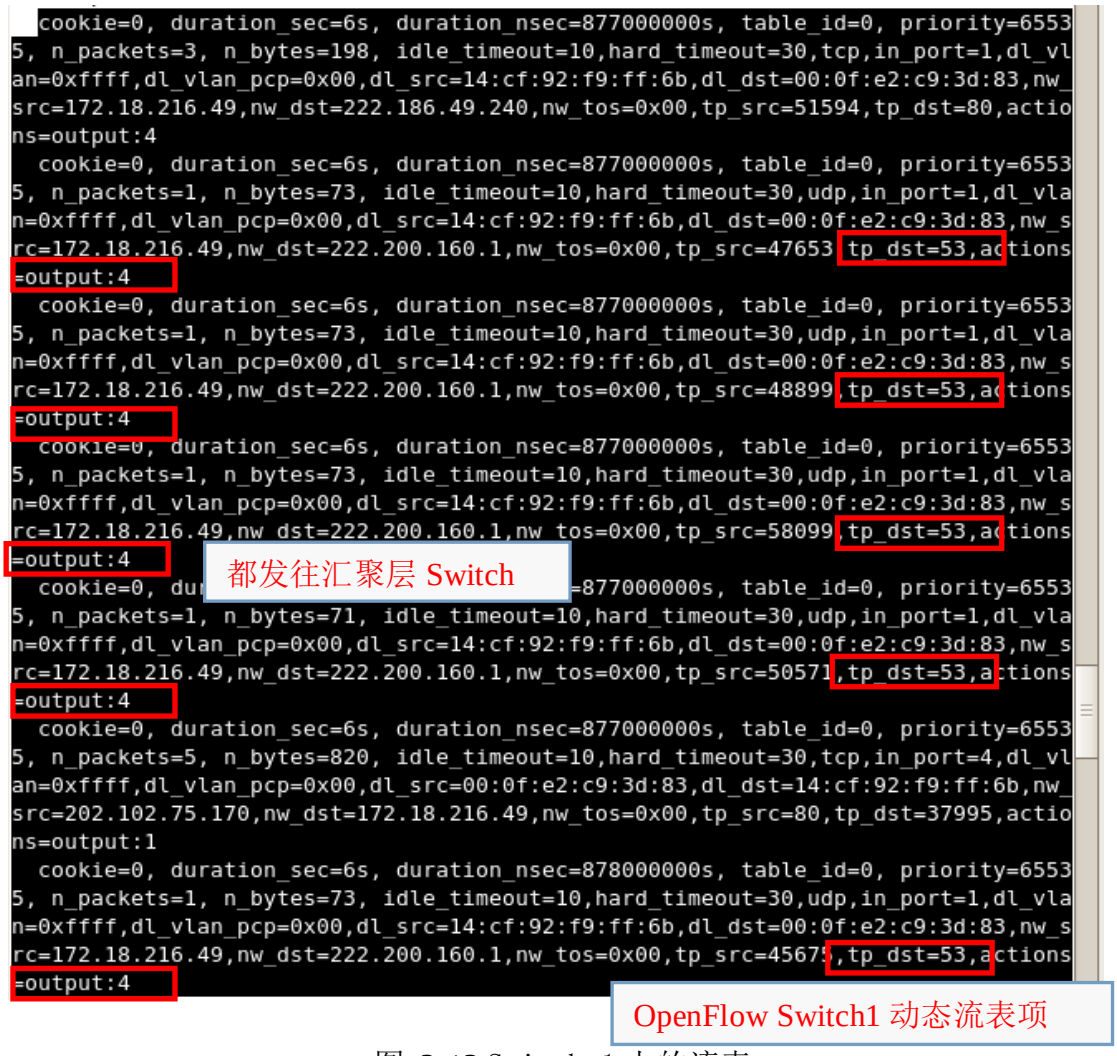

图 3-13 Switch 1 上的流表

PC-B 上 Switch2 上流表如图 3-14 所示：

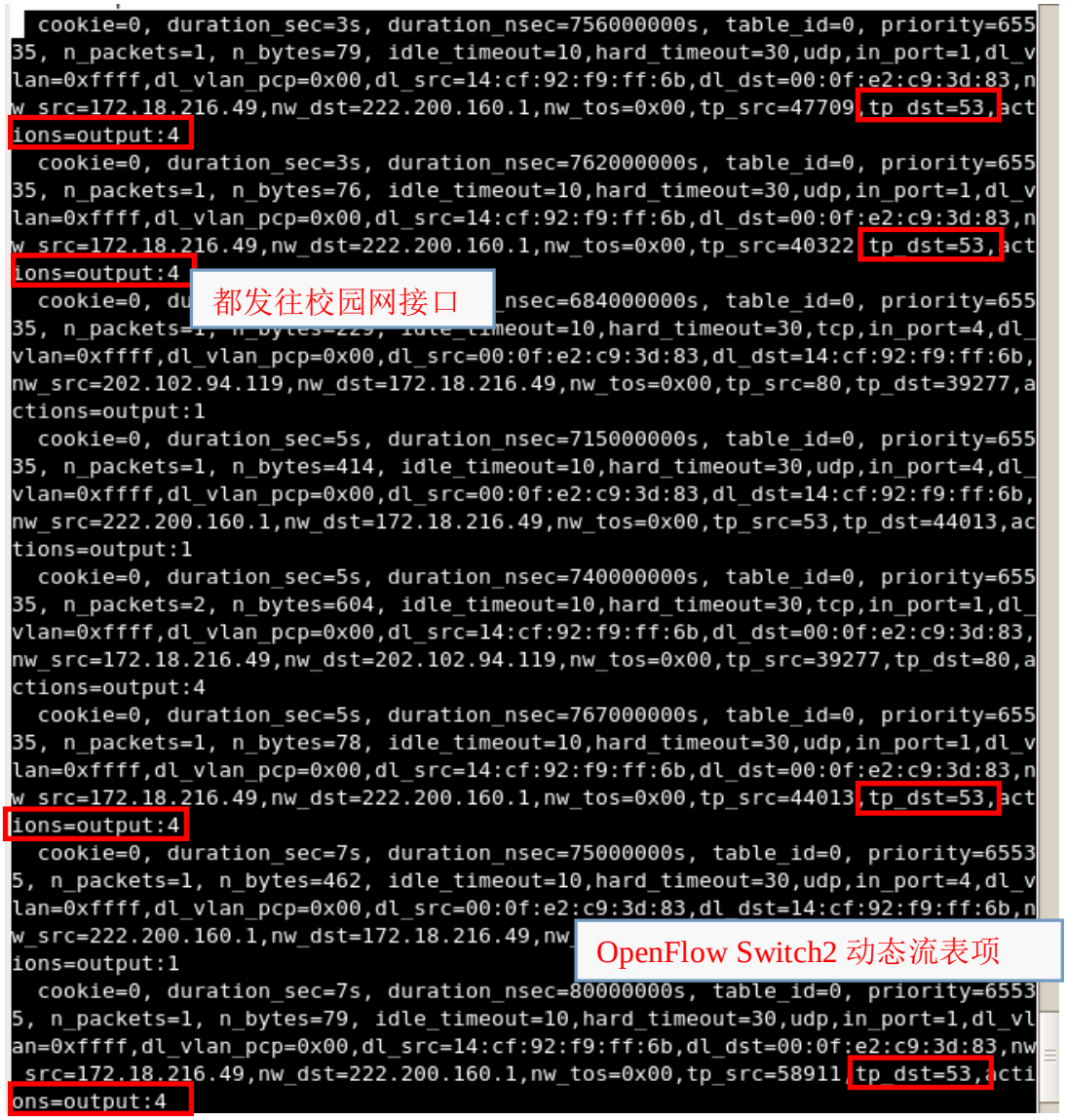

图 3-14　　Switch 2 上的流表

SDN 控制器白名单网站访问屏幕显示如图 3-15 所示：

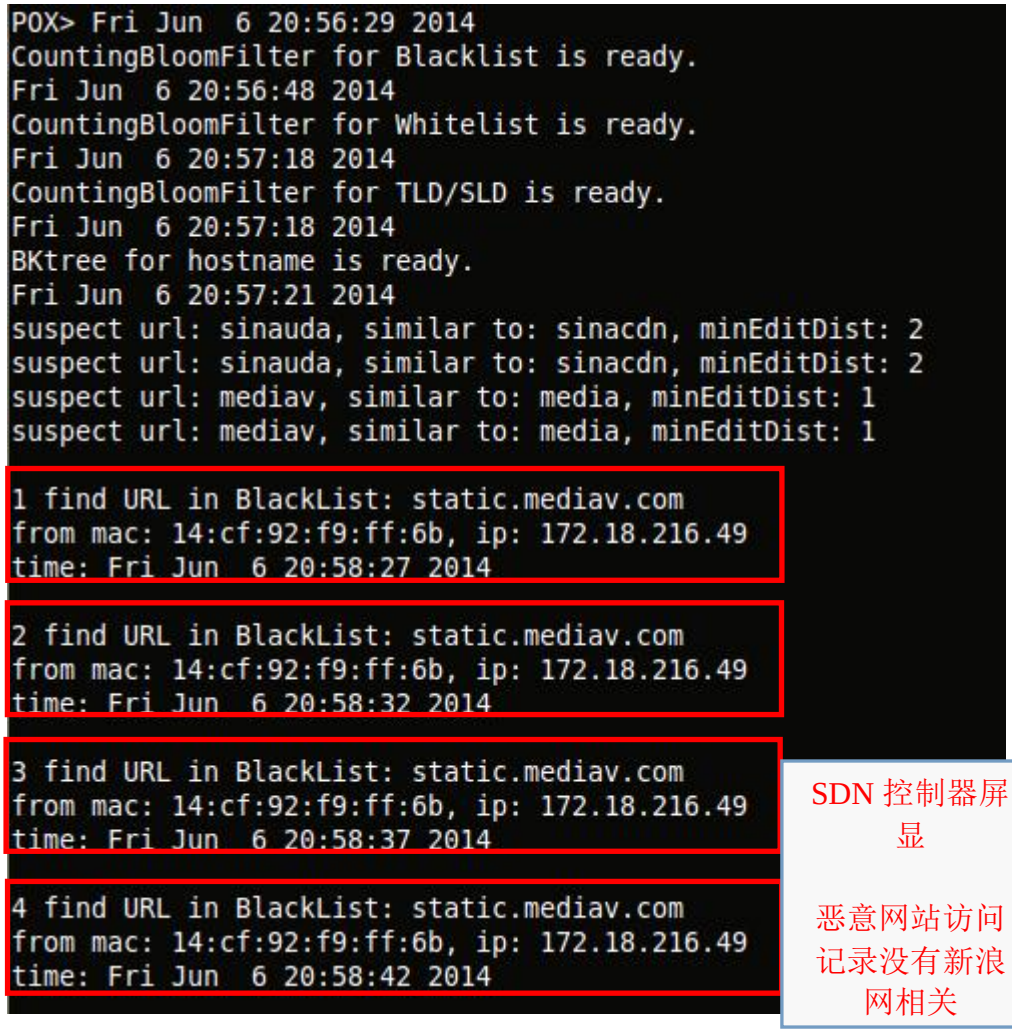

图 3-15 SDN 控制器白名单网站访问屏幕显示

3.2.3 实验结果及其现象（二）——恶意网站访问

客户端 PC-0，加载 dnsv1.py 模块，其 DNS 域名请求名单为“blacklist.txt”，PC 客户端对恶意网站进行 DNS 访问请求，通过 Wireshark 观察 Eth1 端口数据包情况，如图 3-16 所示：

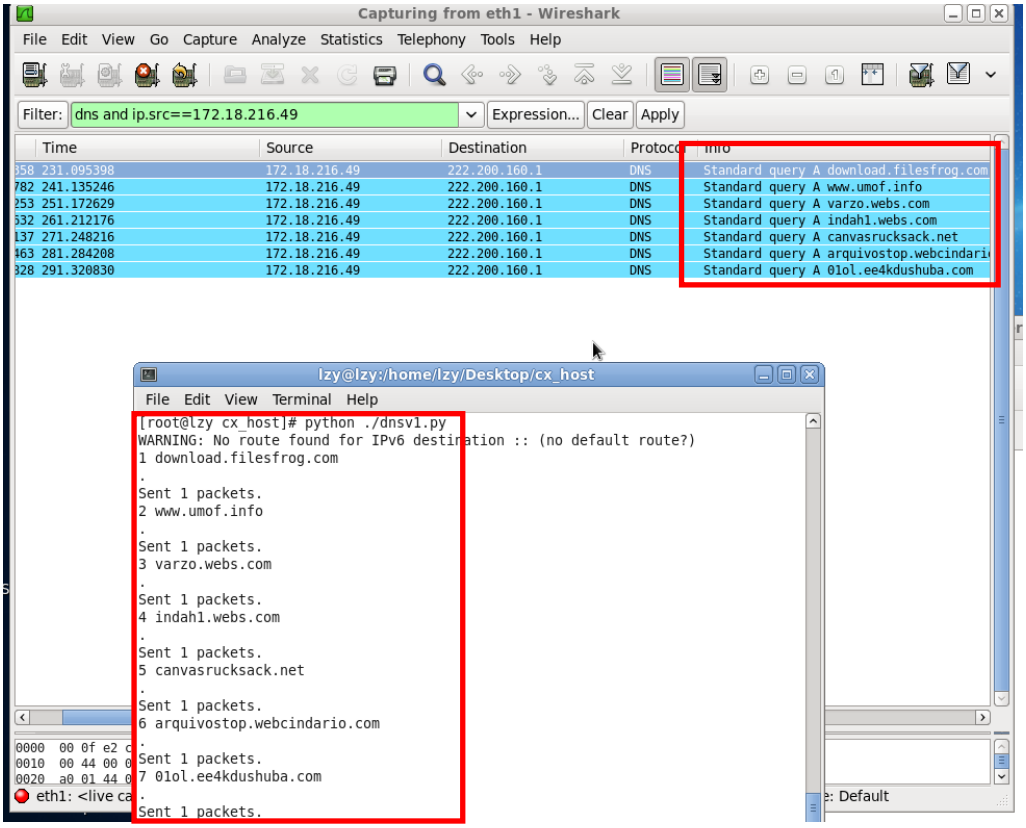

图 3-16 Wireshark 观察 Eth1 端口数据包情况

SDN 控制器检测到 OpenFlow 交换机送上来的 DNS 数据包，对数据包的解析可以发现是“download.filesfrog.com”，“www.umof.info”，“varzo.webs.com”，“indah1.webs.com”，“canvasrucksack.net”等网站的 DNS 访问请求，查询白名单并没有匹配，于是通过查询由黑名单生成的 Counting Bloom Filter，判断其为恶意网站，拒绝该网站的 DNS 解析请求服务。

SDN 控制器恶意网站访问警报信息如图 3-17 所示：

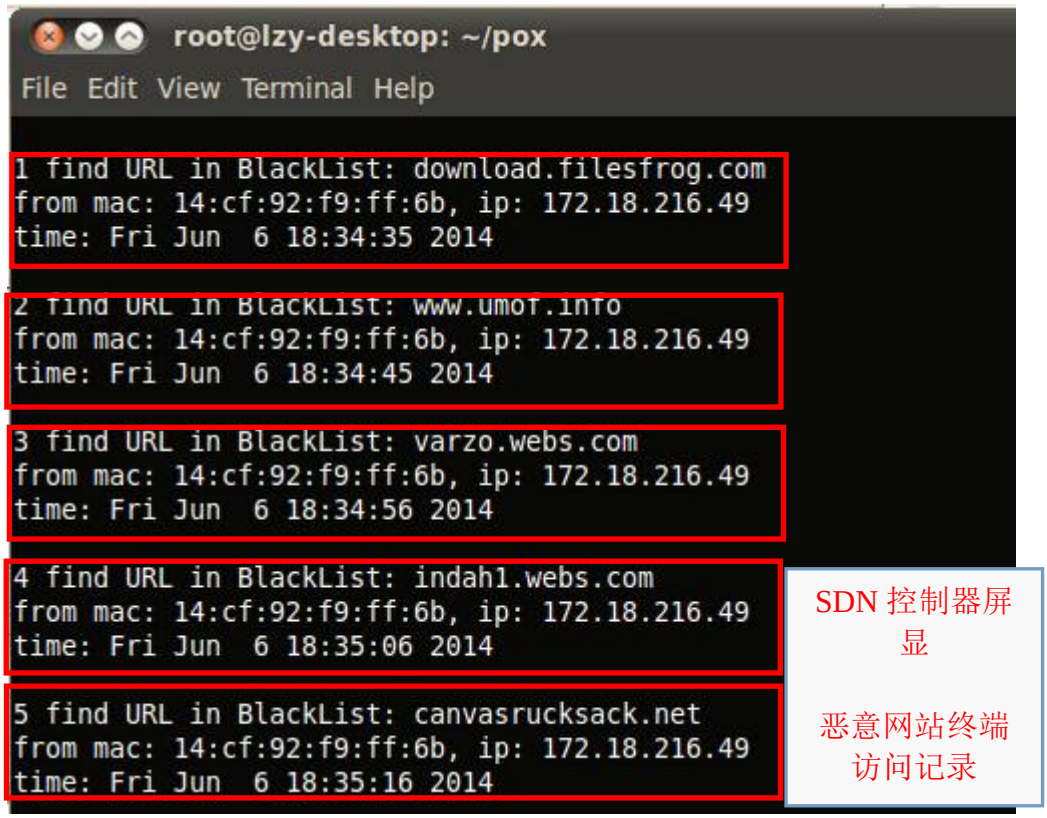

图 3-17 SDN 控制器恶意网站访问警报信息

查看生成的 firewall.log 可以发现对应的恶意网站访问记录，如图 3-18 所示：

图 3-18 firewall.log 对应的恶意网站访问记录

3.2.4 实验结果及其现象（三）——可疑网站访问

客户端 PC-0，加载 dnsv1.py 模块，其 DNS 域名请求名单为“Sus.txt”，模拟 PC 客户端对可疑网站进行 DNS 访问请求，通过 Wireshark 观察 Eth1 端口数据包情况，如图 3-19 所示：

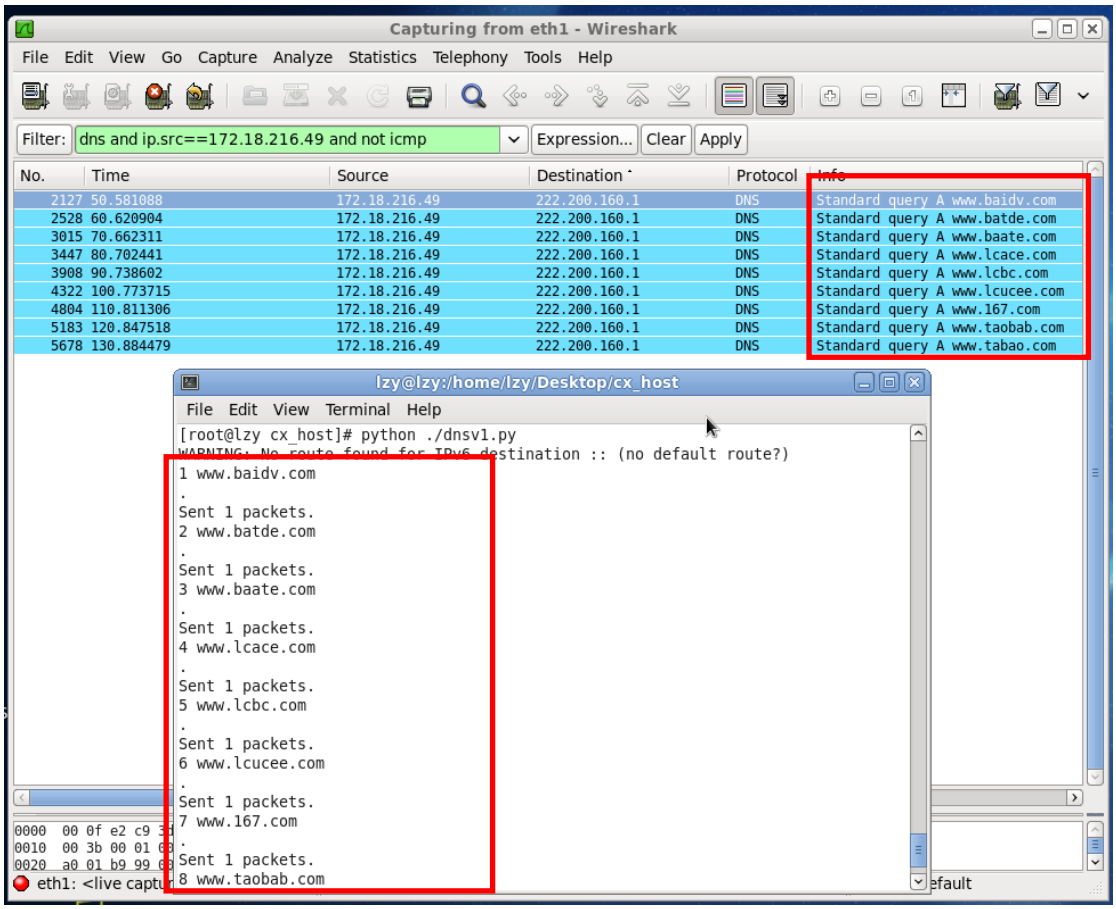

图 3-19 Wireshark 观察 Eth1 端口数据包情况

表 3-1 Sus.txt

DNS 请求域名	主机名	参考主机名	编辑距离
www.baidv.com	baidv	baidu	1
www.batde.com	batde	baidu	1
www.baate.com	baate	baidu	3
www.lcace.com	lcace	lacare	2
www.lcbc.com	lcbc	icbc	1
www.lcucee.com	lcucee	icbc	4
www.167.com	167	163	1
www.taobab.com	taobab	taobao	1
www.tabao.com	tabao	taobao	1

SDN 控制器检测到 OpenFlow 交换机送上来的 DNS 数据包，对数据包的解

析可以发现是“www.baidv.com”，“www.batde.com”,“www.baate.com”,“www.lcace.com”,“www.lcbc.com”,“www.lcucee.com”“www.167.com”,“www.taobab.com”,“www.tabao.com”等编辑距离在 1 到 4 之间网站的 DNS 访问请求，查询白名单并没有匹配，于是查询黑名单生成的 CountingBloomFilter，也无法匹配，进入可疑域名智能检测模块，检测结果如图 3-20 所示：

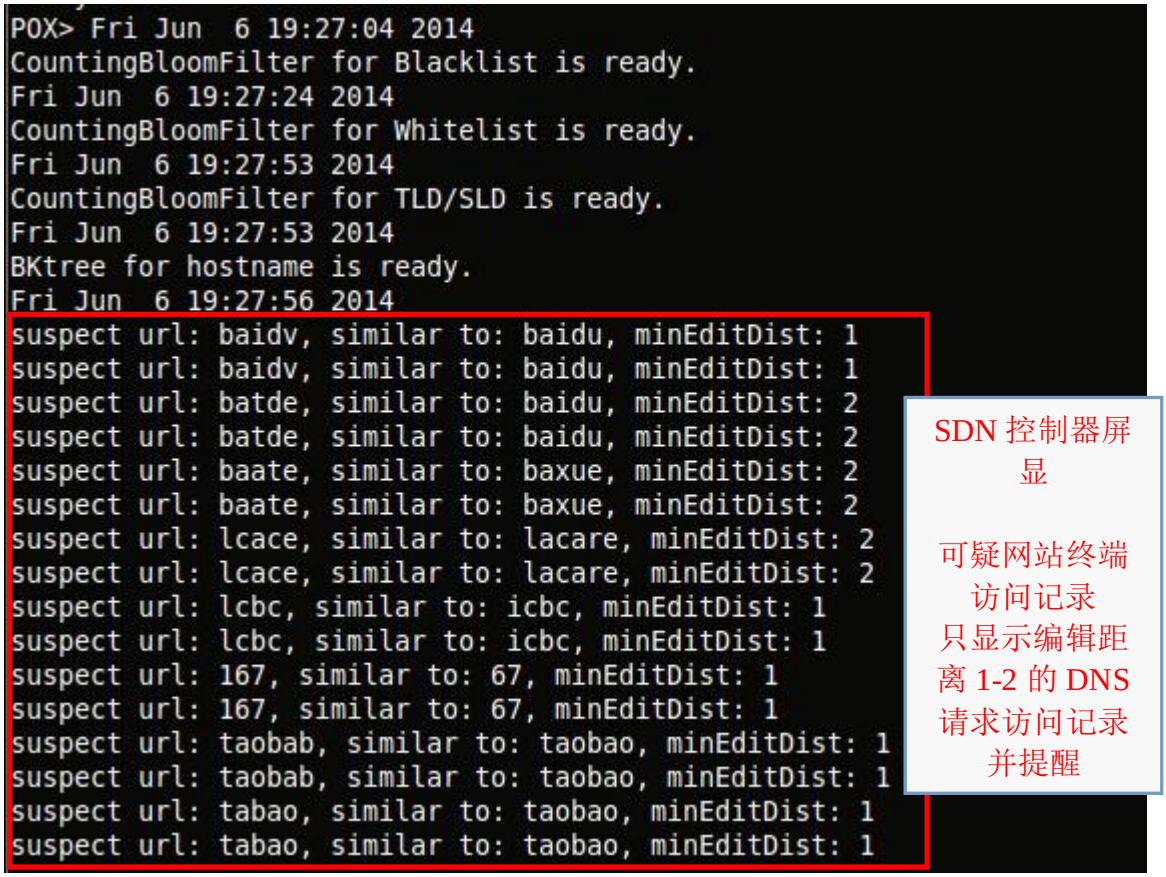

图 3-20 可疑域名智能检测模块检测结果

查看生成的 suspectURL.log 可以发现对应的可疑网站访问记录，如图 3-21 所示：

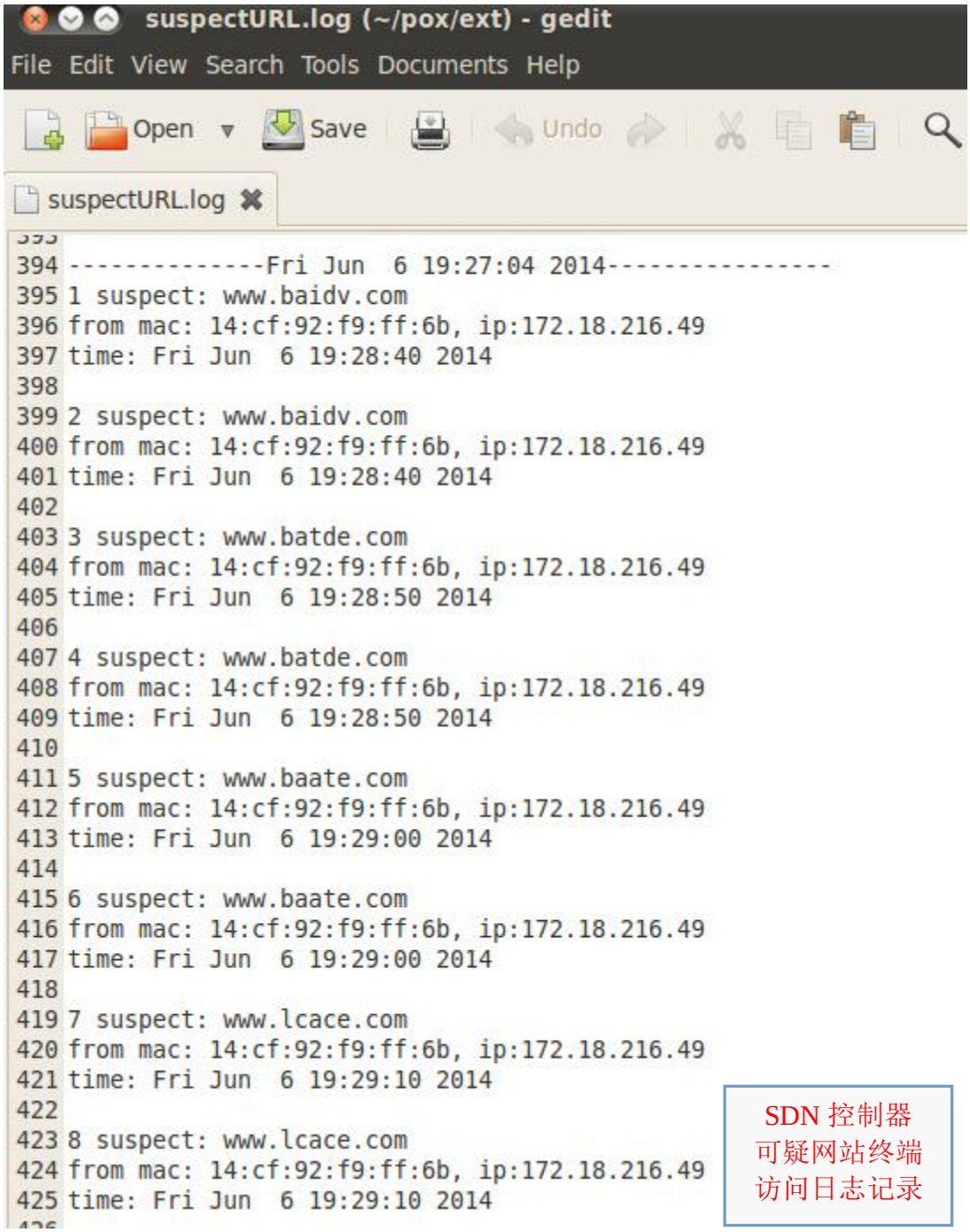

图 3-21 suspectURL.log 对应的可疑网站访问记录

3.2.5 实验结果及其现象（四）——黑白名单列表的增减

为了满足终端用户可以对白黑名单域名以及 TLD\SLD 列表的增减需求，我们增加了名单修改模块。如 2.2.1 节表 2-1 所示。由于名单修改模块对各名单修改原理一致，因此，仅以黑名单列表增加删减为例。

（一）黑名单列表的增加

客户端 PC-0，加载 dnsv1.py 模块，其 DNS 域名请求名单为“BLadd.txt”，通过设置 opcode 字段数值为 10，模拟通过 PC 客户端发送黑名单列表增加的 DNS 请求，并且通过 Wireshark 观察 Eth1 端口数据包情况，如图 3-22 所示：

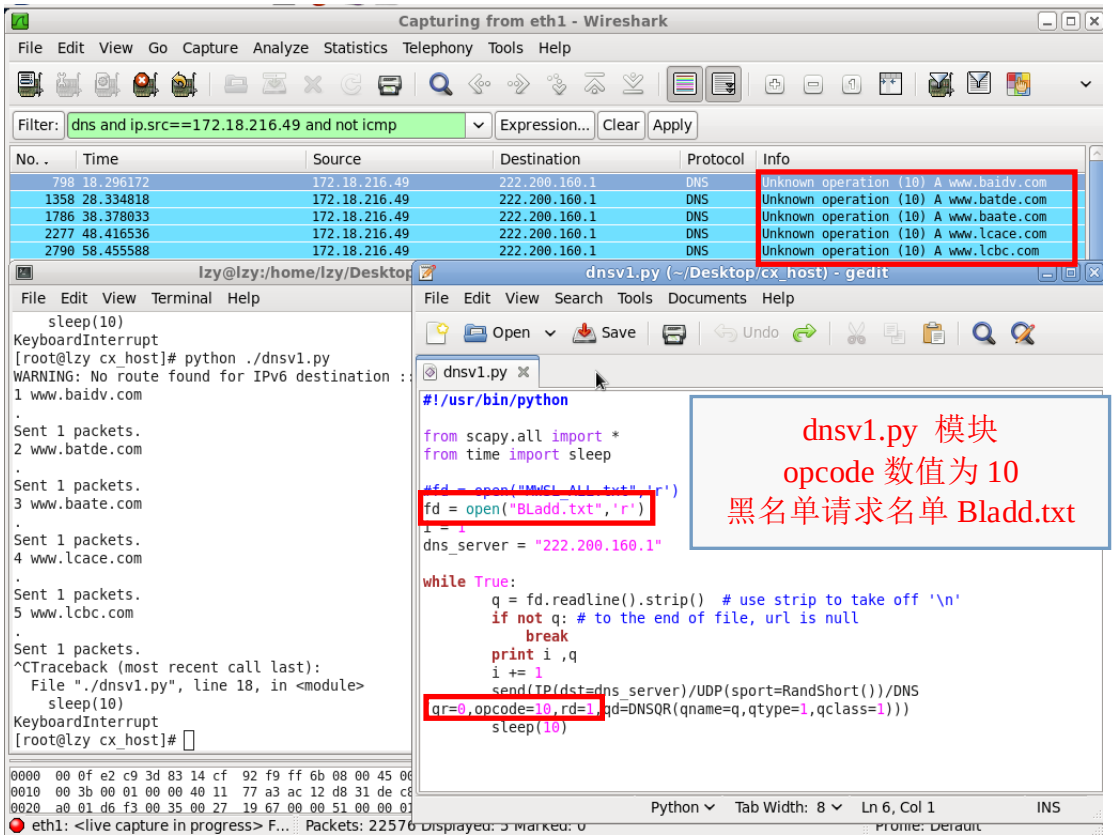

图 3-22 Wireshark 观察 Eth1 端口数据包情况

黑名单域名增加的结果如图 3-23 所示：

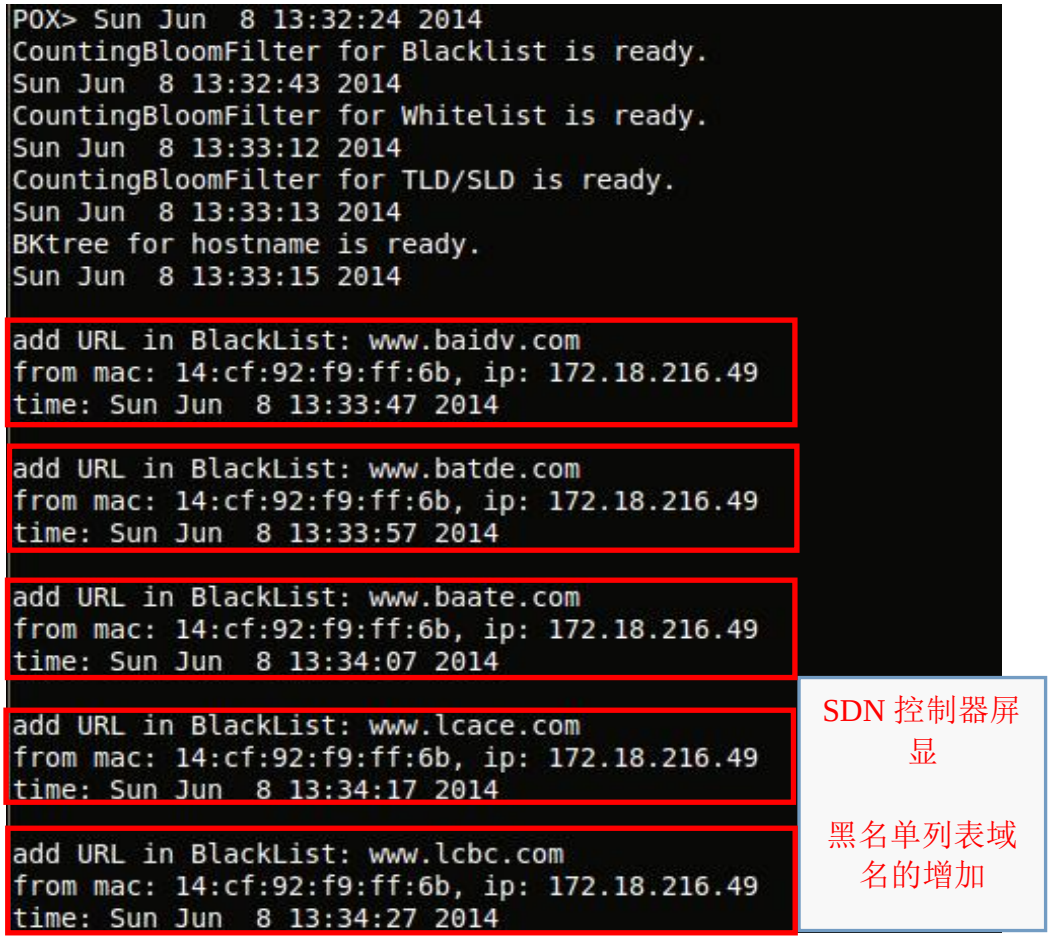

图 3-23 黑名单列表域名增加检测结果

查看生成的addBLURL.log可以发现黑名单列表域名的增加记录，如图3-24所示：

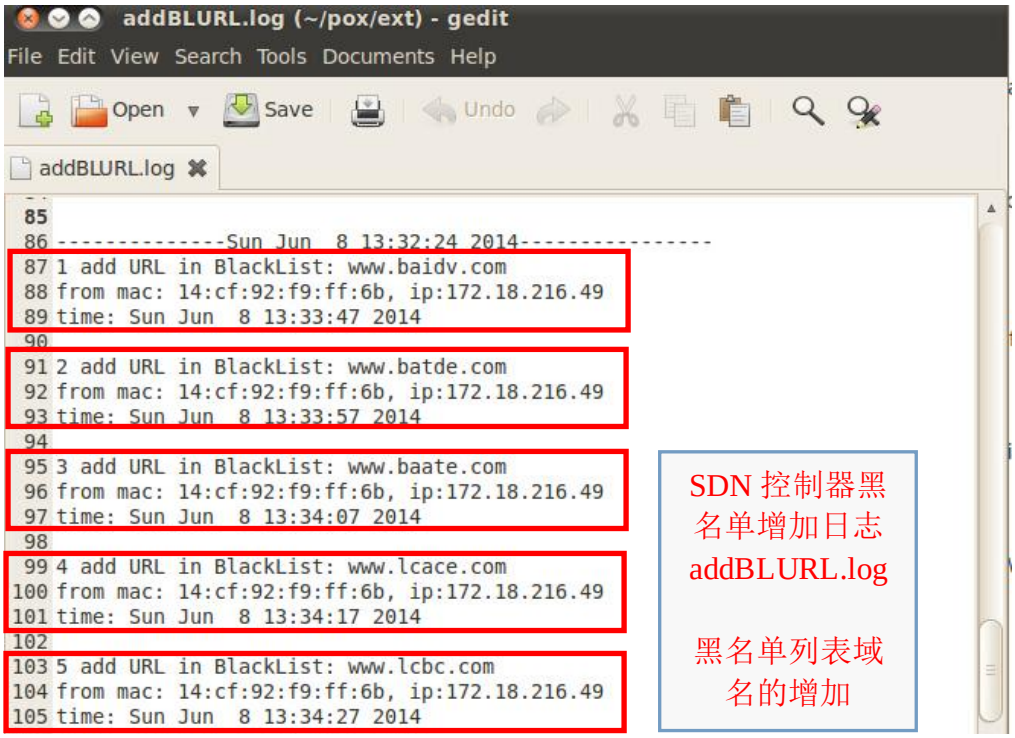

图 3-24 addBLURL.log 对应的黑名单列表域名增加记录

（二）黑名单列表的删减

客户端 PC-0，加载 dnsv1.py 模块，其 DNS 域名请求名单为“BLdel.txt”，通过设置 opcode 字段数值为 10，模拟通过 PC 客户端发送黑名单列表删减的 DNS 请求，并且通过 Wireshark 观察 Eth1 端口数据包情况，如图 3-25 所示：

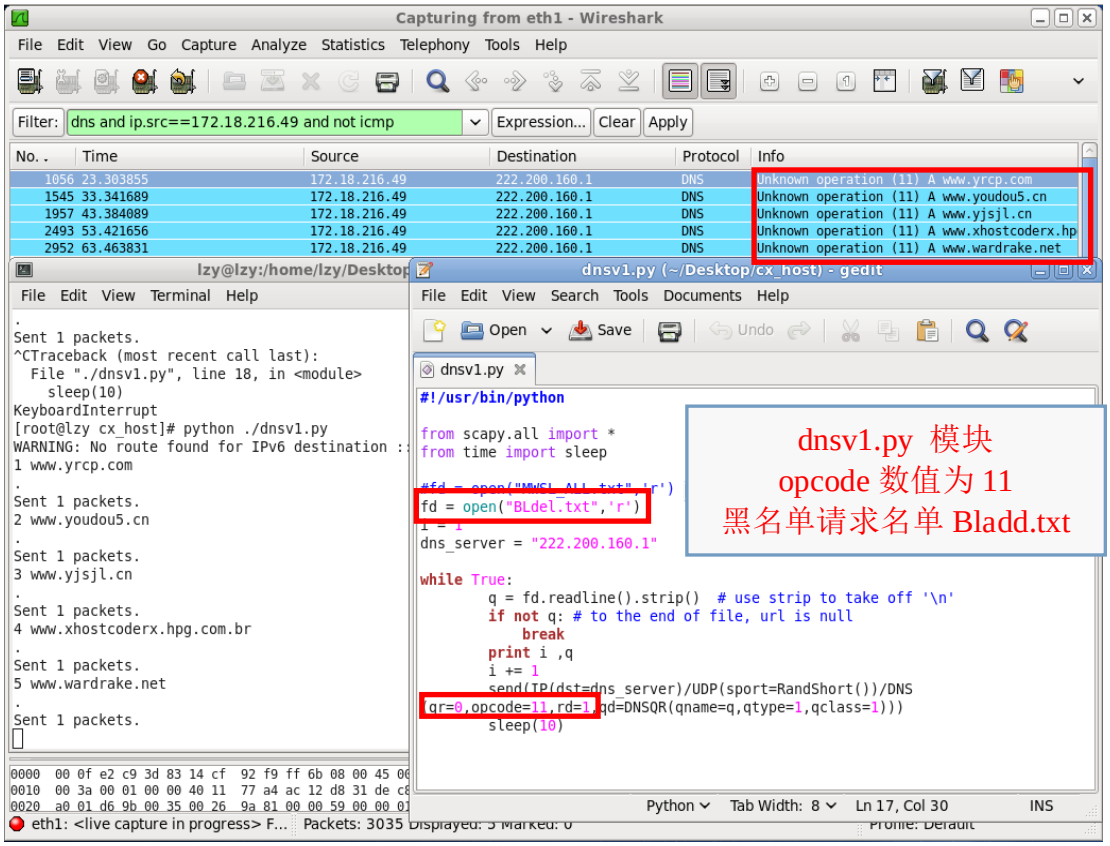

图 3-25 Wireshark 观察 Eth1 端口数据包情况

黑名单域名删除的结果如图 3-26 所示，查看生成的 deleteBLURL.log 可以发现黑名单列表域名的增加记录，如图 3-27 所示：

图 3-26 黑名单列表域名删减检测结果

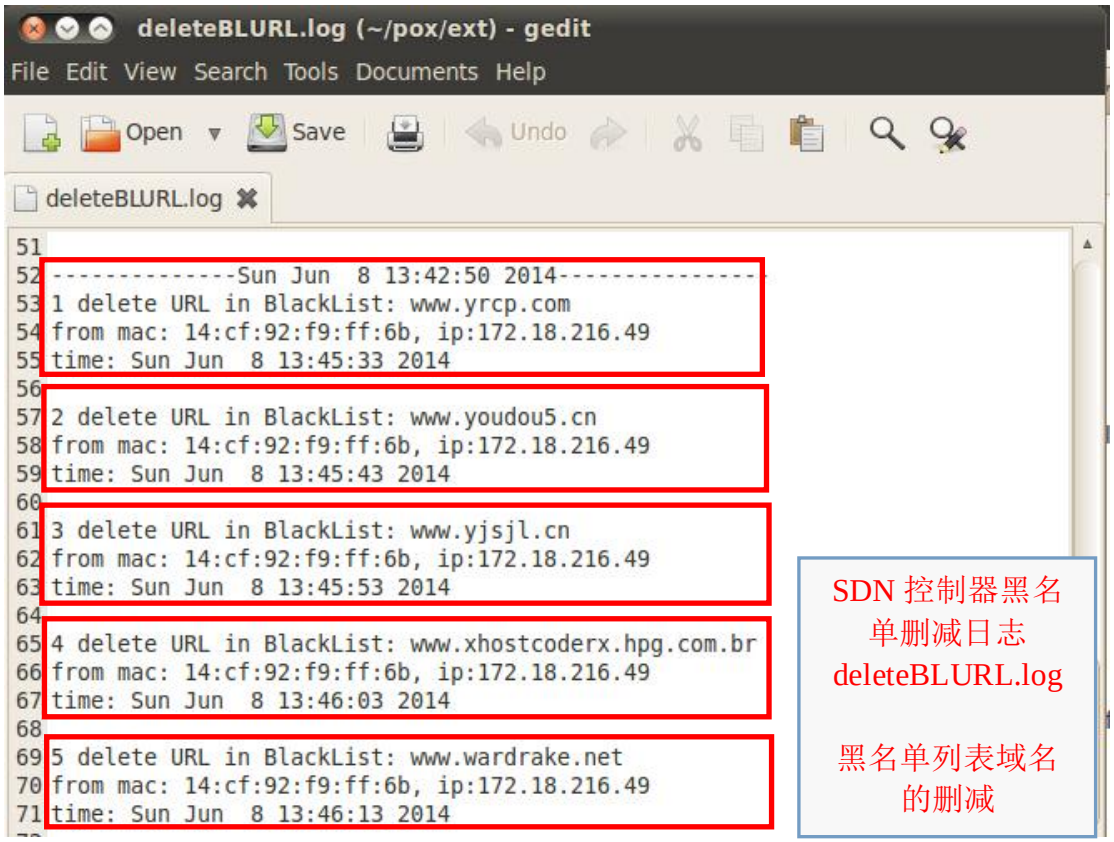

图 3-27 deleteBLURL.log 对应的黑名单列表域名删减记录

3.2.6 实验结果及其现象（五）——TLD/SLD 名单列表的增减

（一）TLD/SLD 名单列表的增加

客户端 PC-0，加载 dnsv1.py 模块，其 DNS 域名请求名单为“TLDadd.txt”，通过设置 opcode 字段数值为 14，模拟通过 PC 客户端发送 TLD/SLD 列表增加的 DNS 请求，并且通过 Wireshark 观察 Eth1 端口数据包情况，如图 3-28 所示：

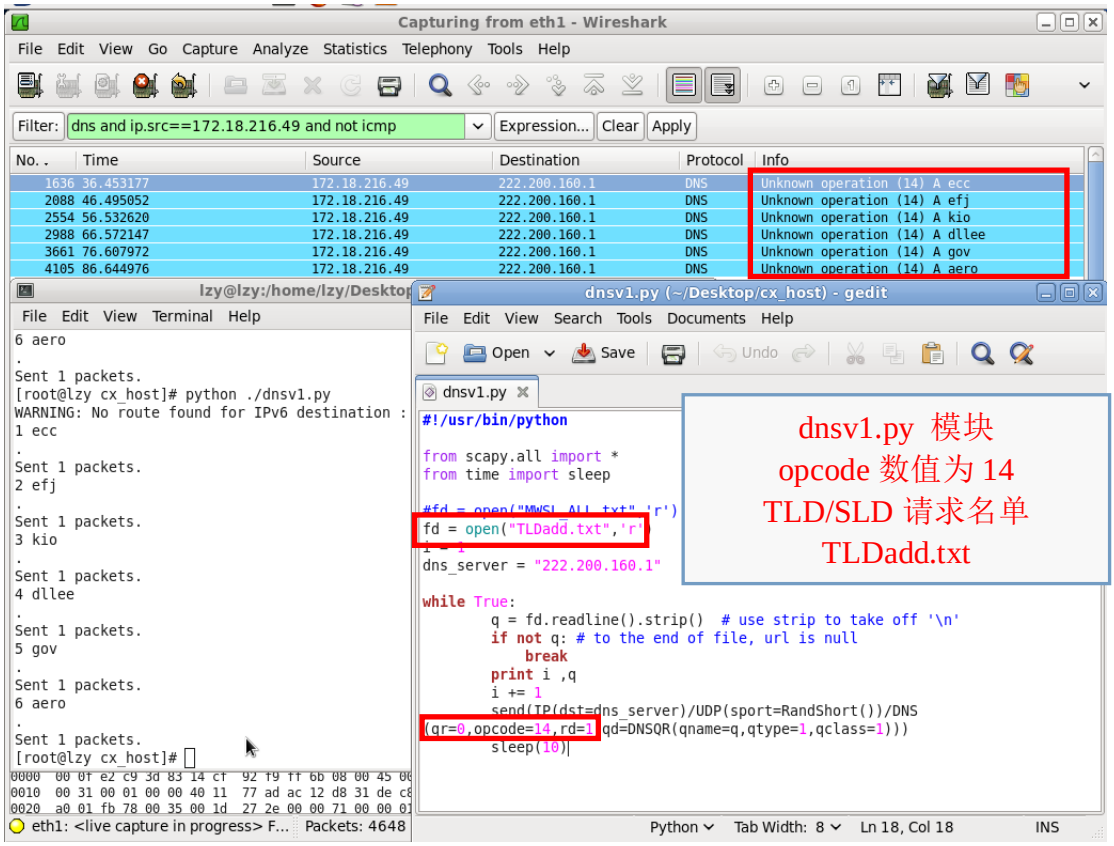

图 3-28 Wireshark 观察 Eth1 端口数据包情况

TLD/SLD 名单列表增加检测结果如图 3-29 所示：

可以发现，对于 TLD 名单里面没有的 TLD/SLD，会增加名单列表，而对于已经存在的 TLD/SLD，则会报错，提醒用户已经存在该 TLD/SLD。

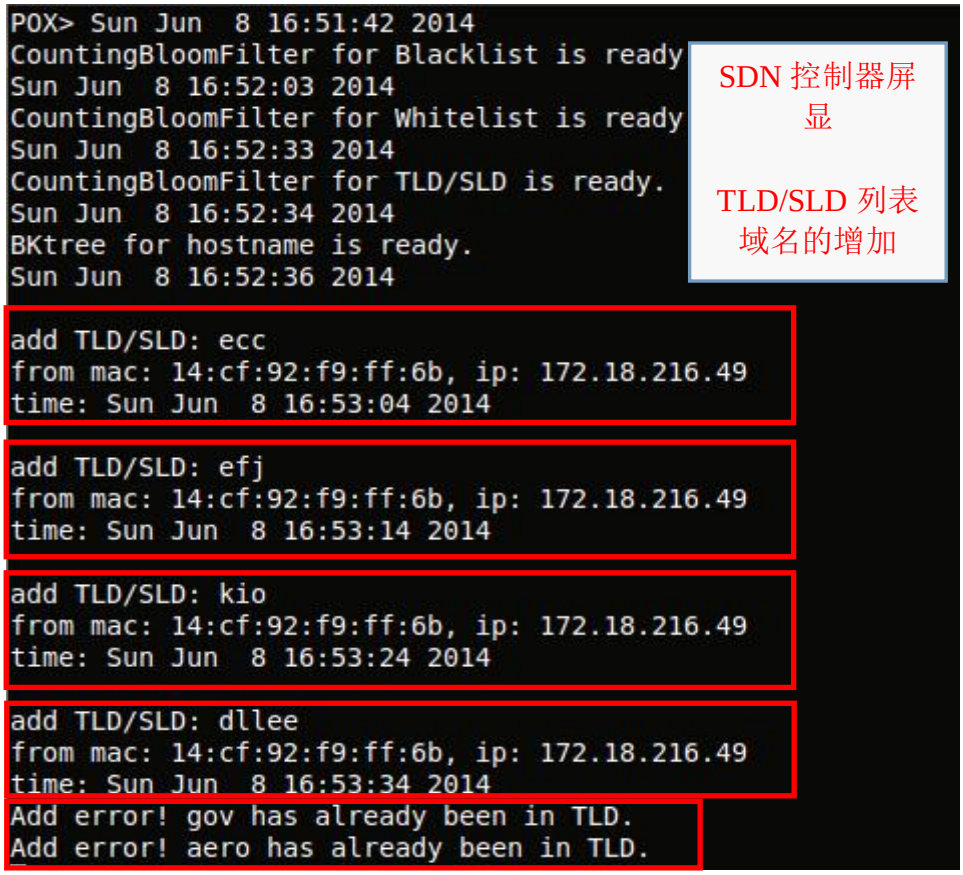

图 3-29 TLD/SLD 名单列表增加检测结果

查看生成的 addTLD.log 可以发现 TLD/SLD 名单列表域名的增加记录，如图 3-30 所示：

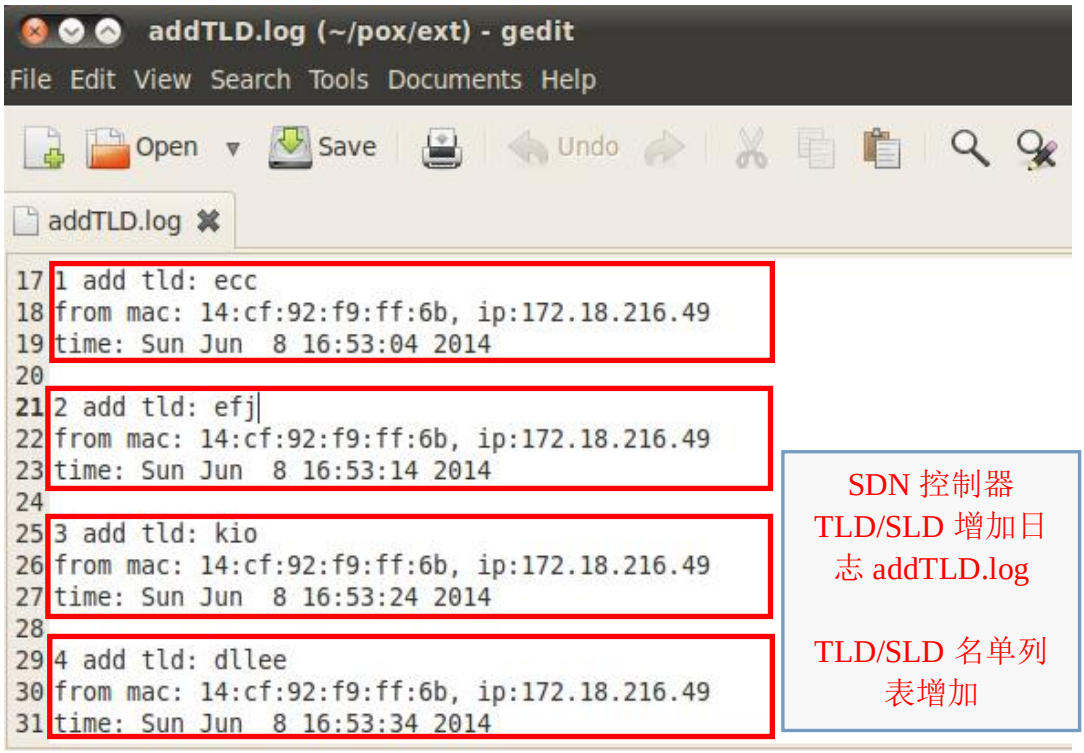

图 3-30 addTLD.log 对应的 TLD/SLD 名单列表域名增加记录

（二）TLD/SLD 名单列表的删减

客户端 PC-0，加载 dnsv1.py 模块，其 DNS 域名请求名单为“TLDdel.txt”，通过设置 opcode 字段数值为 15，模拟通过 PC 客户端发送 TLD/SLD 列表删减的 DNS 请求，并且通过 Wireshark 观察 Eth1 端口数据包情况，如图 3-25 所示：

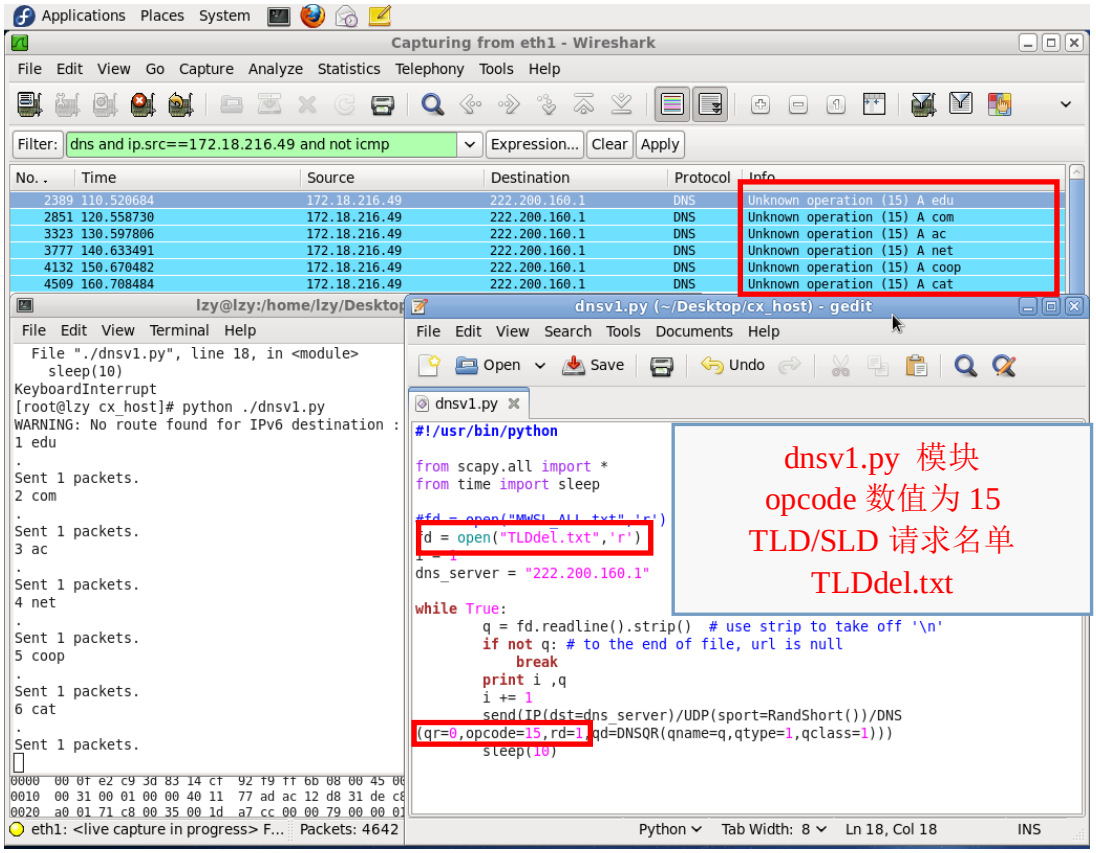

图 3-31 Wireshark 观察 Eth1 端口数据包情况

TLD/SLD 名单列表删减检测结果如图 3-32 所示：

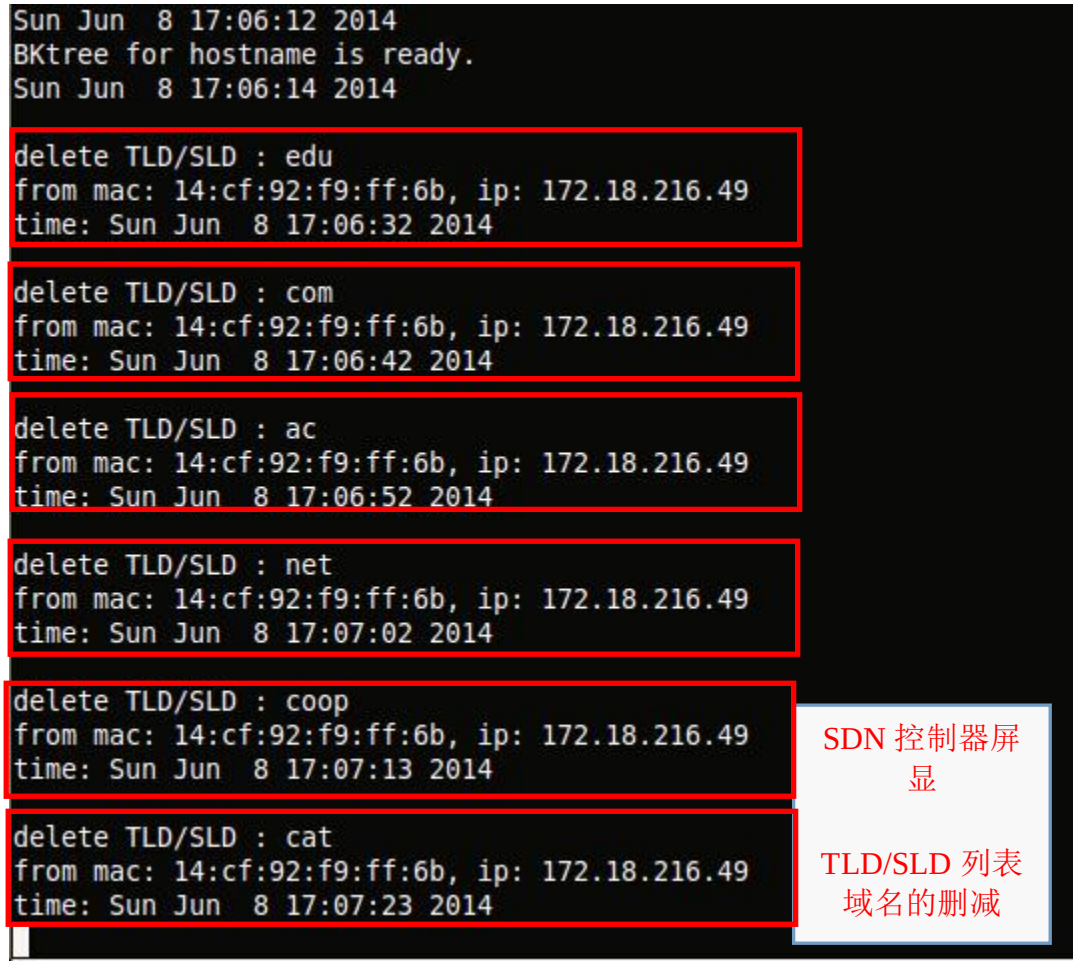

图 3-32 TLD/SLD 名单列表删减检测结果

查看生成的 deleteTLD.log 可以发现 TLD/SLD 名单列表域名的删除记录，如图 3-33 所示：

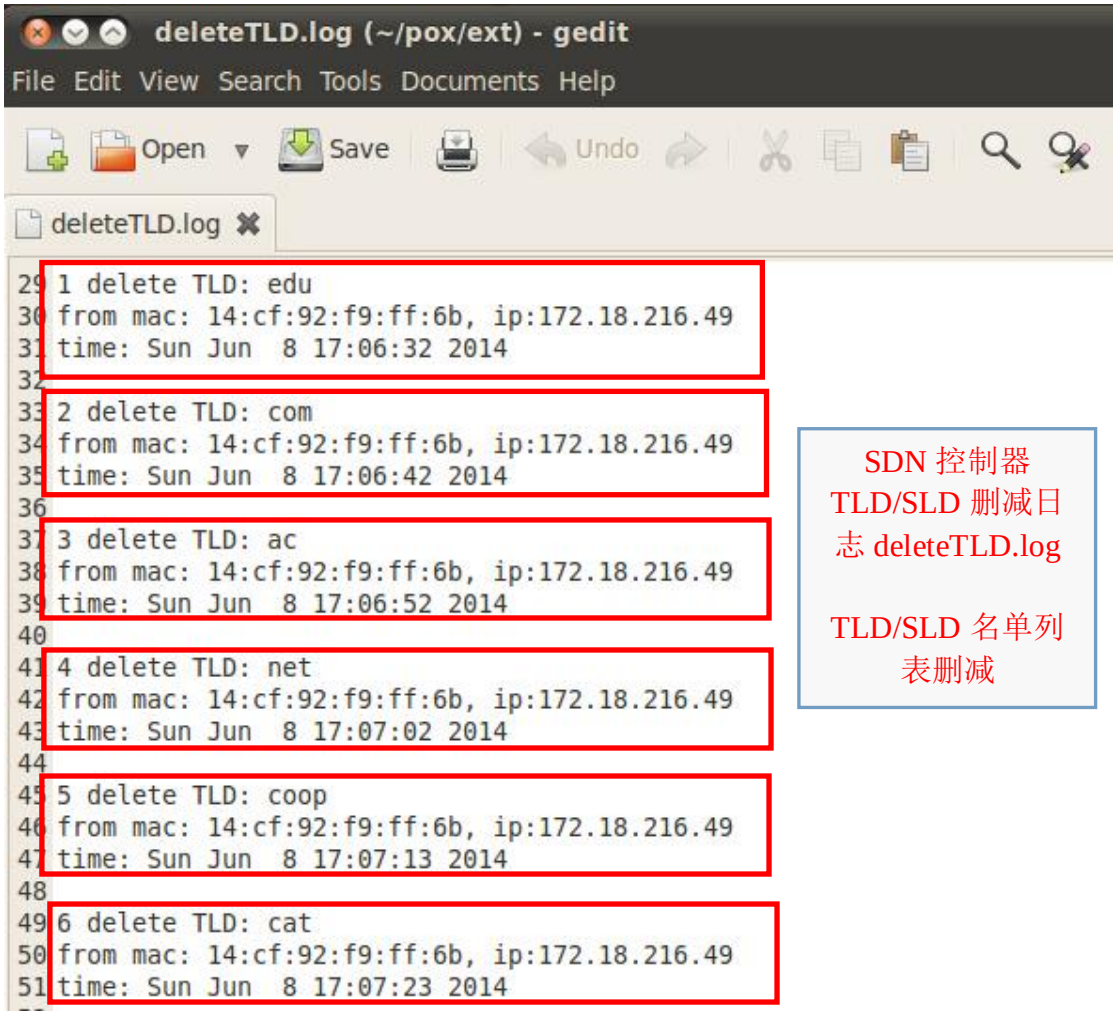

图 3-33 addTLD.log 对应的 TLD/SLD 名单列表域名删减记录

参考文献

[1] [美]. Thomas D. Nadeau, K. Gray.著, and 单. 毕军, 张绍宇, 姚广 译, *软件定义网络：SDN 与 OpenFlow 解析*: 人民邮电出版社, 2014.4.

[2] 赵慧玲，雷葆华，王峰，王茜，王和宇，解云鹏，刘圆，史凡, *SDN 核心技术剖析和实战指南*: 电子工业出版社, 2013.9.

[3] "NetFPGA," http://netfpga.org/.

[4] 与非门科技(北京)有限公司. "NetFPGA 技术中文研讨社区," http://netfpga.eefocus.com/.

[5] "POX Wiki," https://openflow.stanford.edu/display/ONL/POX%20Wiki.

[6] A. Z. M. Broder, M., "Network applications of Bloom filters: A survey," *nternet Mathematics*, vol. 1, no. 4, pp. 485-509, 2003.

[7] 云满山头. "python 布隆过滤器"; http://palydawn.blog.163.com/blog/static/182969056201210260470485/.

[8] "Scapy," http://www.secdev.org/projects/scapy/.

[9] L. Fan, P. Cao, J. Almeida *et al.*, "Summary cache: A scalable wide-area Web cache sharing protocol," *Ieee-Acm Transactions on Networking*, vol. 8, no. 3, pp. 281-293, Jun, 2000.

[10] IANA. "Domain Name System (DNS) Parameters," http://www.iana.org/assignments/dns-parameters/dns-parameters.xhtml.

[11] as_. "编辑距离及编辑距离算法," http://www.cnblogs.com/biyeymyhjob/archive/2012/09/28/2707343.html.

[12] 郑礼雄，李青山，李素科，袁春阳，"基于域名信息的钓鱼 URL 探测，" *计算机工程*, no. 10, pp. 108-110, 2012.

[13] W. A. Burkhard, and R. M. Keller, "Some approaches to best-match file searching," *Commun. ACM*, vol. 16, no. 4, pp. 230-236, 1973.

[14] 行云. "字符串哈希函数," http://www.cnblogs.com/uvsjoh/archive/2012/03/27/2420120.html.

[15] Matrix67. "编辑距离、拼写检查与度量空间：一个有趣的数据结构," http://www.matrix67.com/blog/archives/333.

附录 1 BloomFilte.py

```
#########################################################################
BloomFilte.py    Bloom Filter 数据结构
#########################################################################
#_*_coding:utf_8_
import cmath
from BitVector import BitVector

class BloomFilter(object):
    def __init__(self, error_rate, elementNum):
        #计算所需要的 bit 数
        self.bit_num = -1 * elementNum * cmath.log(error_rate) / (cmath.log(2.0))

        #四字节对齐
        self.bit_num = self.align_4byte(self.bit_num.real)

        #分配内存
        self.bit_array = BitVector(size=self.bit_num)

        #计算 hash 函数个数, only effected by error_rate
        self.hash_num = cmath.log(2) * self.bit_num / elementNum
        self.hash_num = self.hash_num.real

        #向上取整
        self.hash_num = int(self.hash_num) + 1

        #产生 hash 函数种子
        self.hash_seeds = self.generate_hashseeds(self.hash_num)

    def insert_element(self, element):
        for seed in self.hash_seeds:
            hash_val = self.hash_element(element, seed)
            self.bit_array[hash_val] = 1     #设置相应的比特位

    #检查元素是否存在，存在返回 true，否则返回 false
    def has_element(self, element):
        for seed in self.hash_seeds:
            hash_val = self.hash_element(element, seed) #calculate hash value
            if self.bit_array[hash_val] == 0:     #查看值
                return False
        return True
```

```
#内存对齐
def align_4byte(self, bit_num):
    num = int(bit_num / 32)
    num = 32 * (num + 1)
    return num

#产生 hash 函数种子,hash_num 个素数
def generate_hashseeds(self, hash_num):
    count = 0
    #连续两个种子的最小差值
    gap = 50

    hash_seeds = []
    for index in xrange(hash_num):   #初始化 hash 种子为 0
        hash_seeds.append(0)
    for index in xrange(10, 10000):
        max_num = int(cmath.sqrt(1.0 * index).real)
        flag = 1
        for num in xrange(2, max_num):
            if index % num == 0:
                flag = 0
                break

        if flag == 1:
            #连续两个 hash 种子的差值要大才行
            if count > 0 and (index - hash_seeds[count - 1]) < gap:
                continue
            hash_seeds[count] = index
            count = count + 1

        if count == hash_num:
            break
    return hash_seeds

def hash_element(self, element, seed): # BKDR hash for string
    hash_val = 1
    for ch in str(element):
        hash_val = hash_val * seed + ord(ch)

    hash_val = abs(hash_val)   #取绝对值
    hash_val = hash_val % self.bit_num   #取模，防越界
    return hash_val
```

附录 2 CountingBloomFilte.py

```
#########################################################################
CountingBloomFilte.py    CountingBloom Filter 数据结构
#########################################################################

#_*_coding:utf_8_
import cmath
from BitVector import BitVector

class CountingBloomFilter(object):
      def __init__(self, error_rate, elementNum):
            #计算所需要的 bit 数
            self.bit_num = (-1) * elementNum * cmath.log(error_rate) /
(cmath.log(2.0))

            #四字节对齐
            self.bit_num = self.align_4byte(self.bit_num.real)

      #chxfan: CBF needs 4 times that BF need
      self.bit_num = 4*self.bit_num
      print 'bit_num = %d' % self.bit_num

            #分配内存
            self.bit_array = BitVector(size=self.bit_num)

            #计算 hash 函数个数, only effected by error_rate
            self.hash_num = cmath.log(2) * self.bit_num / elementNum
            self.hash_num = self.hash_num.real

            #向上取整
            self.hash_num = int(self.hash_num) + 1
            print 'hash_num = %d' % self.hash_num

            #产生 hash 函数种子
            self.hash_seeds = self.generate_hashseeds(self.hash_num)
      #for s in self.hash_seeds:
      #     print s

       def insert_element(self, element):
      tmp_c = 0
            for seed in self.hash_seeds:
```

```
                hash_val = self.hash_element(element, seed)
                #self.bit_array[hash_val] = 1     #设置相应的 4 比特位
            tmp = self.bit_array[4*hash_val:4*hash_val+4]
            if tmp.intValue() <= 14:
            self.bit_array[4*hash_val:4*hash_val+4]       =       BitVector(size=4,intVal=
tmp.intValue() + 1)
            if tmp.intValue() > 1:
            tmp_c += 1
            if tmp_c == self.hash_num:
                    print "The new added element (%s) is not able to be deleted." %
(element)

        #检查元素是否存在，存在返回 true，否则返回 false
        def has_element(self, element):
            for seed in self.hash_seeds:
                hash_val = self.hash_element(element, seed) #calculate hash value
            tmp =    self.bit_array[4*hash_val:4*hash_val+4]
                #if self.bit_array[hash_val] == 0:     #查看值
            if tmp.intValue() == 0:     #查看值
                        return False
            return True

        #删除某个元素
        def delete_element(self, element):
            for seed in self.hash_seeds:
                hash_val = self.hash_element(element, seed)
                tmp =    self.bit_array[4*hash_val:4*hash_val+4]
                # whether element is in CBF or not, should be check by the call of
has_element by the user. the func in CBF should be as simple as possible.
            #if tmp.intValue() == 0:     #查看值
            #print "delete error! No such element: %s." % element
                        # return
            # else:
            self.bit_array[4*hash_val:4*hash_val+4]       =       BitVector(size=4,intVal=
tmp.intValue() - 1)



        #内存对齐
        def align_4byte(self, bit_num):
            num = int(bit_num / 32)
            num = 32 * (num + 1)
            return num
```

```
#产生 hash 函数种子,hash_num 个素数
def generate_hashseeds(self, hash_num):
    count = 0
    #连续两个种子的最小差值
    gap = 50

    hash_seeds = []
    for index in xrange(hash_num):   #初始化 hash 种子为 0
        hash_seeds.append(0)
    for index in xrange(10, 10000):
        max_num = int(cmath.sqrt(1.0 * index).real)
        flag = 1
        for num in xrange(2, max_num):
            if index % num == 0:
                flag = 0
                break

        if flag == 1:
            #连续两个 hash 种子的差值要大才行
            if count > 0 and (index - hash_seeds[count - 1]) < gap:
                continue
            hash_seeds[count] = index
            count = count + 1

        if count == hash_num:
            break
    return hash_seeds

def hash_element(self, element, seed): # BKDR hash for string
    hash_val = 1
    for ch in str(element):
        hash_val = hash_val * seed + ord(ch)

    hash_val = abs(hash_val)   #取绝对值
    hash_val = hash_val % (self.bit_num/4)   #取模，防越界
    return hash_val
```

附录 3 BKtree.py

```
#############################################################################
BKtree.py    BK-Tree 数据结构
#############################################################################
#!/usr/bin/python

# -*- coding: utf-8 -*-

findOne = False

##==================calculate edit distance====================

def calculateDist(str1, str2):
    len1 = len(str1)
    len2 = len(str2)

    # 初始化矩阵
    diff = [[i+j for j in range(len2 + 1)] for i in range(len1 + 1)]

    for row in range(1,len1+1):
       for col in range(1,len2+1):
          if str1[row-1] == str2[col-1]:
              f = 0
          else:
              f = 1
          diff[row][col]      =      min(diff[row][col-1]+1,      diff[row-1][col]+1,
diff[row-1][col-1]+f)

          #sim = 1 - float(diff[len1][len2])/float(max(len1,len2))  相似度
    return diff[len1][len2] # edit distance

##======================BK tree nod=====================

class TreeNode(object):
    #data: its string
    #children: N>=0 children
    #dist: distance from father node
    def __init__(self,data = 0,children = None,dist = 0):
        self.data = data
        self.children = children
        self.dist = dist
```

```
##-------------get/set-------------------
    def setData(self,data):
        self.data = data

    def getData(self):
        return self.data

    def setChildren(self,children):
        self.children = children

    def getChildren(self):
        return self.children

    def setDist(self,dist):
        self.dist = dist

    def getDist(self):
        return self.dist

##-------------traversal/insert/findMinDist-------------------

    def DFStraversal(self):
        #print self.data, self.dist
        if self.children == None:
            return
        else:
            for i in range(len(self.children)):
                self.children[i].DFStraversal()

    def insert(self,node):
        if(self.data == node.getData()):
            return
            #print "%s is already in BK tree." % self.data
        elif self.children == None: # 无子节点
            node.setDist( calculateDist(self.data, node.getData()) )
            self.children = [node]
        else:    # 当前节点的 data 和插入节点的不同，而且当前节点有子节点
            dist = calculateDist(self.data, node.getData())
            for i in range(len(self.children)):
                if self.children[i].getDist() == dist:
                    self.children[i].insert(node)
                    return
            #所有直接子节点的 dist 都不一样
            node.setDist(dist)
```

```
                self.children.append(node)

        # 寻找 node.data 对应的最小 edit dist 和相应节点的 data
        def findMinDist(self,node,minDist,minData,beta):
                dist = calculateDist(self.data, node.getData())
                #print self.data,dist
                # 更新到目前为止（当前节点也算进去）的最小 dist 与对应的 hostname
                if dist < minDist:
                        minDist = dist
                        minData = self.data
                #print dist, self.data, node.getData(),minDist,minData

                # 找到相同 data 的节点（dist==0）或者 发现编辑距离的最小差异(dist==1)，则返回，并结束递归循环
                global findOne
                if dist == 0 or dist == 1:
                        findOne = True
                        return [minDist,minData]
                elif self.children == None: # 当前节点没有子节点，则返回
                        return [minDist,minData]
                else: # 与 self.data 不匹配，而且有子节点
                        min_x = max(1, abs(dist - beta))
                        max_x = dist + beta
                        for i in range(len(self.children)):
                                if self.children[i].getDist() >= min_x and self.children[i].getDist() <= max_x:
                                        [minDist,minData] = self.children[i].findMinDist(node,minDist,minData,beta)
                                        if findOne:
                                                break
                        return [minDist,minData]

##=========================BK tree=========================

class BKtree(object):
        def __init__(self,root = None):
                self.root = root

        def setRoot(self,root):
                self.root = root

        def getRoot(self):
                return self.root
```

```
    def isEmpty(self):
        if self.root == None:
            return True
        else:
            return False

    def DFStraversal(self):
        if self.isEmpty():
            print "empty BK tree"
            return
        else:
            self.root.DFStraversal()

    def insert(self,node):
        if self.isEmpty():
            self.setRoot(node)
        else:
            self.root.insert(node)

    def findMinDist(self,s,beta):
        if self.isEmpty():
            print "empty BK tree"
            return [-1.-1]
        else:
            global findOne
            findOne = False
            node = TreeNode(s)
            return  self.root.findMinDist(  node,calculateDist(self.root.getData(),s),
self.root.getData(), beta )
```