# Software Define Network Lab4

**实验目的：**

理解网络故障的普遍性

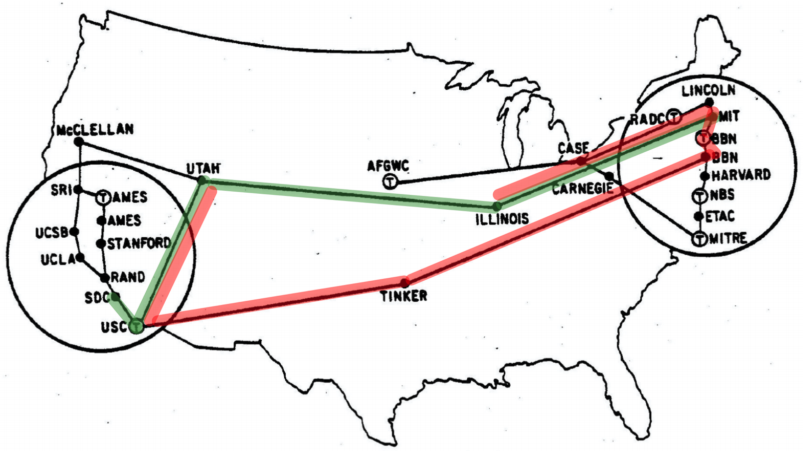
熟悉网络验证工具VeriFlow的原理

掌握VeriFlow的故障检测方法

**实验环境：**

Windows 10, VMware Workstation Pro, Ubuntu

**问题背景：**

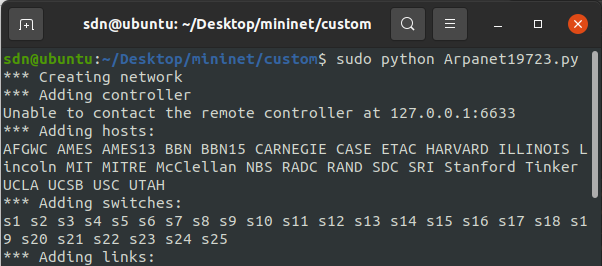


TINKER处建立了一个流量分析中心，为了保证UTAH和ILLINOIS之间的流量可以在TINKER进行分析，你下发了图中的红色路径。你的同事Bob接到了另外一个需求，要求建立从SDC到MIT跳数最少的路径，即图中的绿色路径。不同的路径需求来自不同的用户，没有经过协调，产生了一个转发环路。请你运行VeriFlow工具，对上述两条转发路径进行检查。

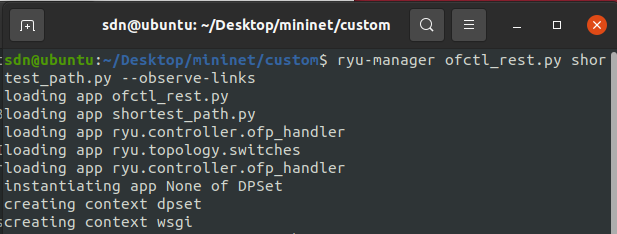
**实验过程：**

1. 示例：如何生成转发环路？

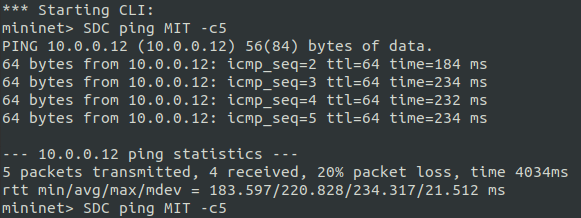
在命令行终端内输入sudo python Arpanet19723.py，启动网络拓扑。



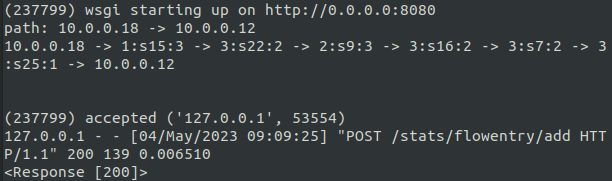
在命令行终端内输入ryu-manager ofctl\_rest.py shortest\_path.py --observe-links，启动最短路径的控制程序。



在mininet命令行内输入SDC ping MIT -c5，建立SDC与MIT之间的链接。

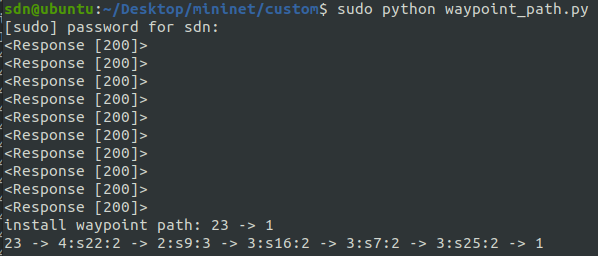


可以看到，此时SDC ping MIT能ping通，两主机之间的连通性没有问题。

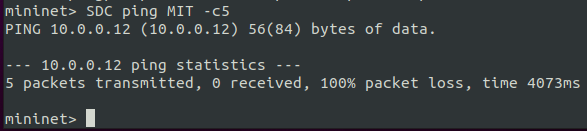


Ryu控制器内也可以看到打印的路径信息与网络信息。

在命令行终端内输入sudo python waypoint\_path.py，下发从UTAH途经TINKER到达ILLINOIS的路径。

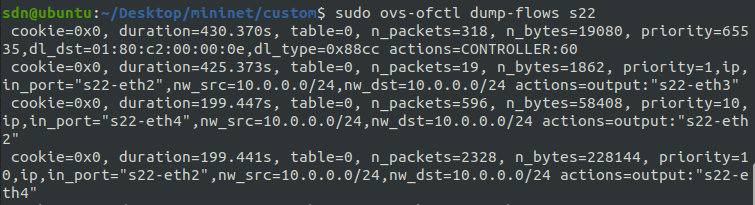


随后在mininet终端内输入SDC ping MIT -c5，测试此时两主机之间的连通性。



可以看到，此时两主机之间无法ping通。

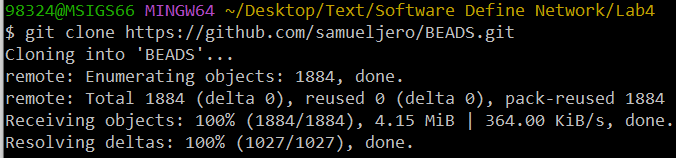
在命令行终端内输入sudo ovs-ofctl dump-flows s22，查看s22交换机的流表。



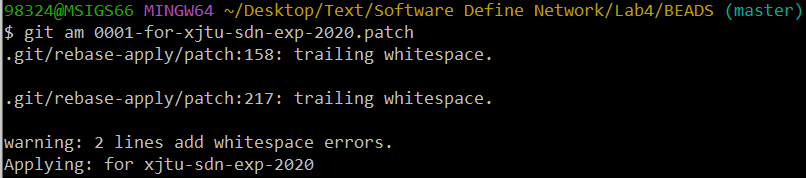
可以看到，匹配某一条流表项的数据包数目异常大，此时拓扑中存在环路。

2. 使用VeriFlow

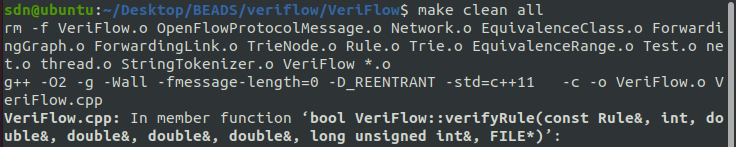
在命令行终端内输入git clone https://github.com/samueljero/BEADS.git



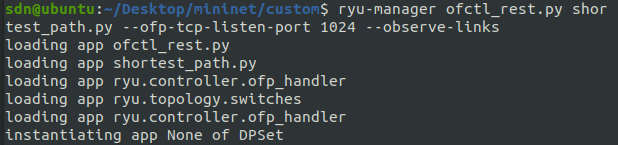
在命令行终端内输入 cd BEADS，切换到/BEADS目录下，然后将补丁文件保存到当前目录下，输入git am 0001-for-xjtu-sdn-exp-2020.patch，打上补丁文件。



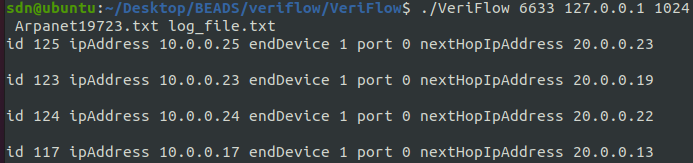
在BEADS/veriflow/VeriFlow目录下，打开命令行终端，输入make clean all，编译VeriFlow。



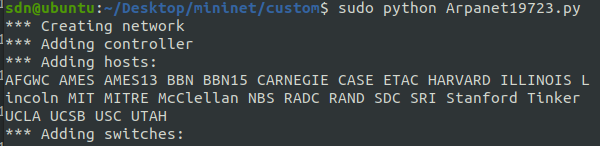
在命令行终端内输入ryu-manager ofctl\_rest.py shortest\_path.py --ofp-tcp-listen-port 1024 --observe-links，在自定义端口开启远程控制器，运行最短路径程序。



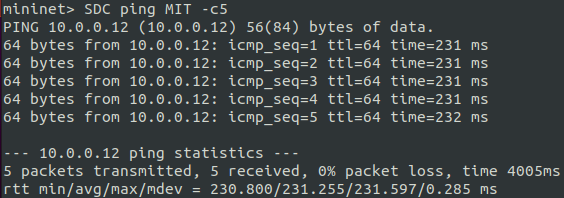
在命令行终端内输入./VeriFlow 6633 127.0.0.1 1024 Arpanet19723.txt log\_file.txt，启动VeriFlow的Proxy模式。



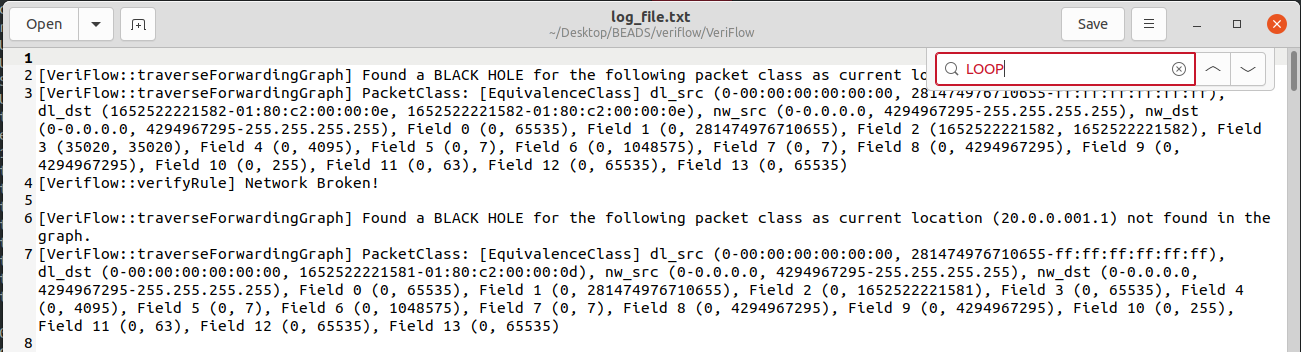
在命令行终端内输入sudo python Arpanet19723.py，启动拓扑。



在mininet命令行内输入SDC ping MIT -c5，建立SDC与MIT之间的链接。

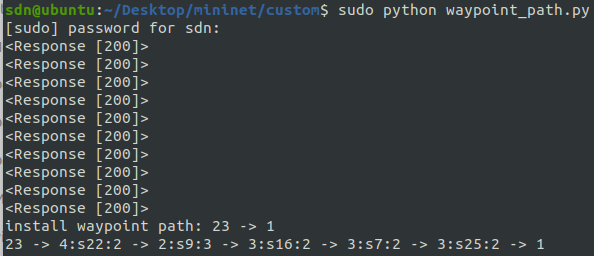


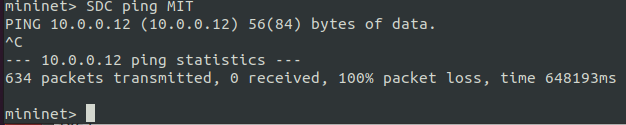
可以看到，此时SDC ping MIT能ping通，两主机之间的连通性没有问题。



查看log\_file.txt可以看到，此时VeriFlow只检测到了黑洞(Black Hole)，没有检测到环路(Loop)。

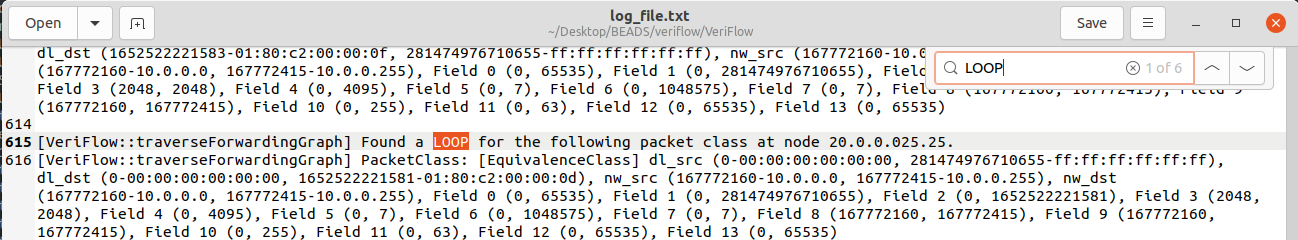
在命令行终端内输入sudo python waypoint\_path.py，下发从UTAH途经TINKER到达ILLINOIS的路径。

 再次在mininet命令行内输入SDC ping MIT，测试两主机连通性。



可以看到，此时两主机之间无法ping通，并且在log文件中观察VeriFlow检测到的环路信息。

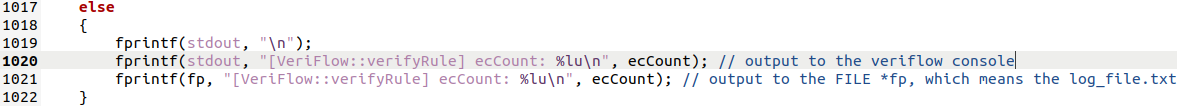
打开BEADS/veriflow/VeriFlow目录下的log\_file.txt文件，可以看到VeriFlow检测到的环路信息。



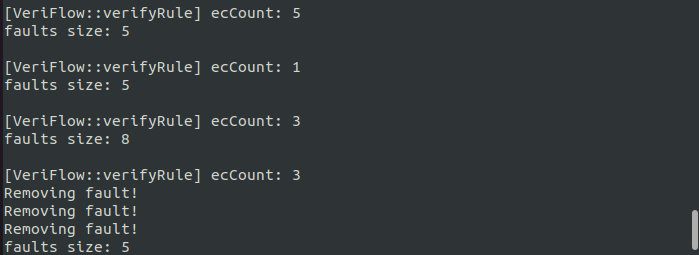
3. 基础实验：EC数目的打印

对每条验证的规则，实验要求输出这条规则所影响的EC数目。

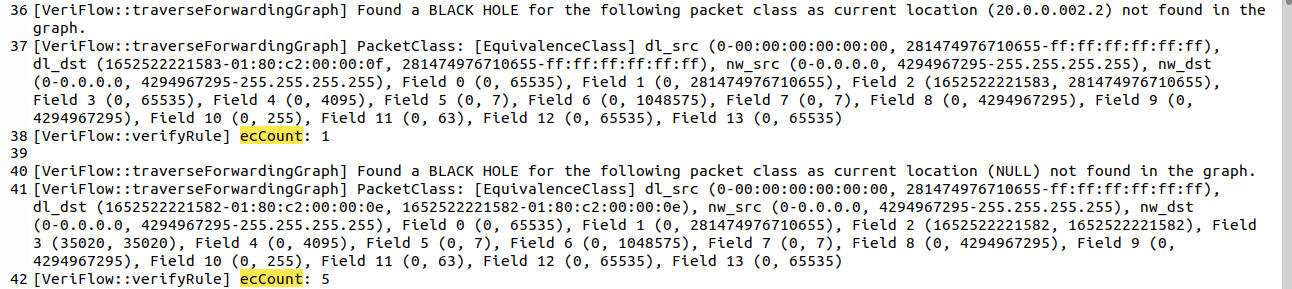
查看源码可以知道，打印EC数目的函数在VeriFlow.cpp中，为VeriFlow::verifyRule()函数。模仿ecCount == 0的代码格式，修改此部分代码如下。

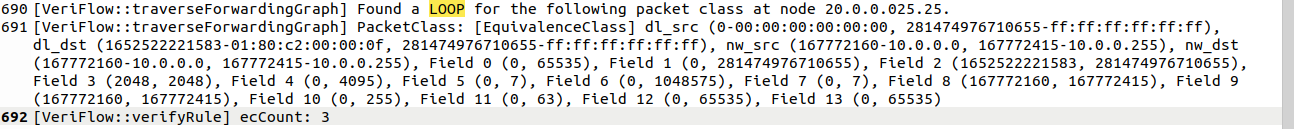


修改完成后重新进行VeriFlow的编译、启动Ryu控制器、启动VeriFlow的Proxy模式、启动拓扑、建立SDC与MIT之间的链接、下发路径、再次测试两主机连通性、查看log\_file.txt。



可以看到，VeriFlow命令行终端打印了受影响的EC数目，一开始在1和5之间变化，稳定后保持为3。





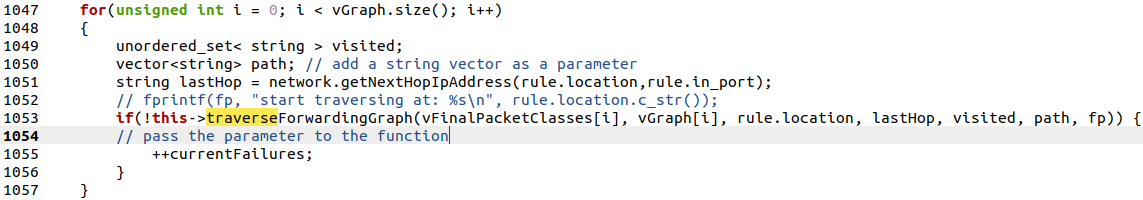
在log\_file.txt中也可以看到相同记录。

4. 基础实验：环路路径的打印

要求打印出环路的信息，包括出现环路的提示信息，EC的基本信息和环路路径上的IP地址。提示：traverseForwardingGraph函数中的visited为unordered\_set，可改成有序的数据结构。

按照实验指导书的提示，traverseForwardingGraph函数的作用是遍历某个特定EC的转发图，验证是否存在环路或黑洞。为了记录环路路径，我们建立一个字符向量vector <string> path用来保存每次的currentLocation。

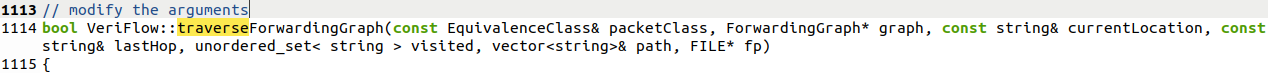
traverseForwardingGraph函数的第一次调用是在VeriFlow::verifyRule函数中，我们建立一个字符向量并修改traverseForwardingGraph函数，将字符向量path作为参数传入函数中。



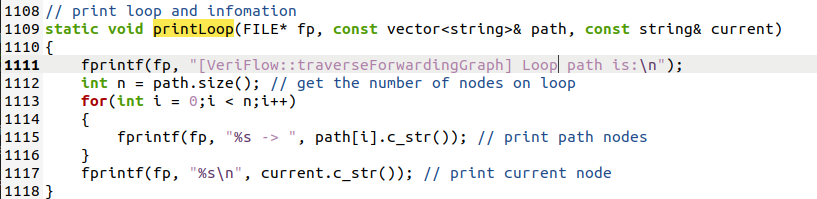
因为我们修改了traverseForwardingGraph函数的定义，所以在VeriFlow.h文件中，也要修改函数对应的声明。



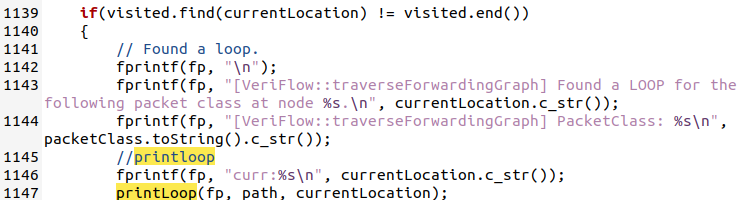
traverseForwardingGraph函数定义在VeriFlow.cpp文件中，我们要修改其函数的定义。



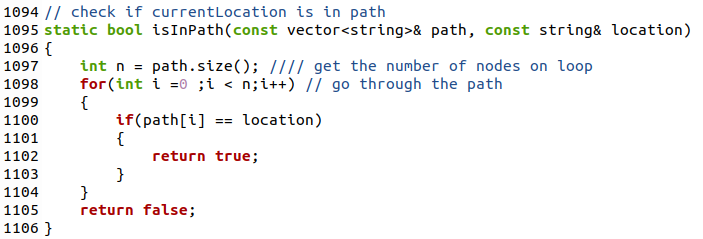
为了打印环路路径，我们定义了一个printLoop函数，用于打印环路路径path上结点的信息与当前结点current的信息。



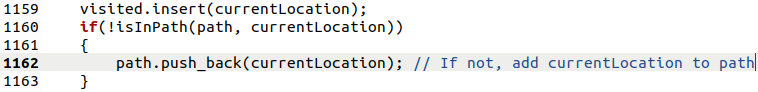
在traverseForwardingGraph函数中，如果存在环路，我们就调用定义的printLoop函数。



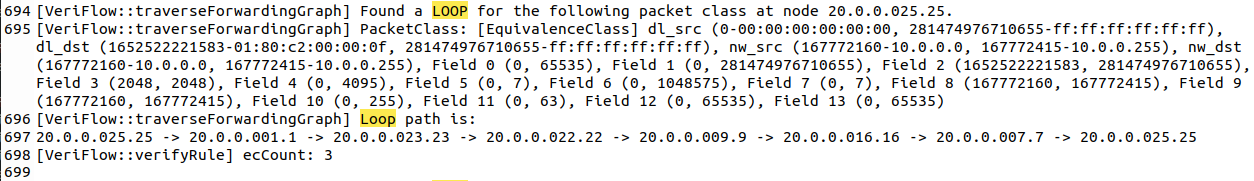
为了判断当前结点currentLocation是否在环路路径path中，我们定义了一个isInPath函数，用于遍历环路路径path，确定当前结点是否在环路路径中。



在traverseForwardingGraph函数中，如果存在环路且当前结点currentLocation不在环路路径path中，就将其添加到path中。



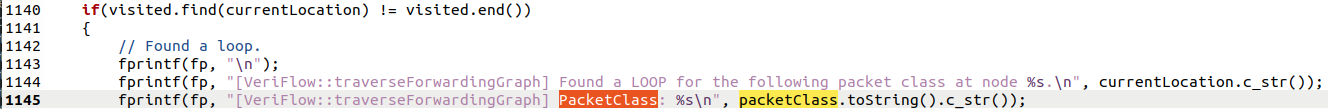
修改完成后重新进行VeriFlow的编译、启动Ryu控制器、启动VeriFlow的Proxy模式、启动拓扑、建立SDC与MIT之间的链接、下发路径、再次测试两主机连通性、查看log\_file.txt。



可以看到，log\_file.txt中打印了环路的信息，包括出现环路的提示信息，EC的基本信息和环路路径上的IP地址。

5. 基础实验：相关数据包信息的打印

EC的基本信息显示为14个域的区间形式，为方便Bob查错，现简化EC信息的表示形式，仅从14个域中提取TCP/IP五元组作为主要信息显示。提示：在环路路径打印的基础上，修改EC的显示格式。

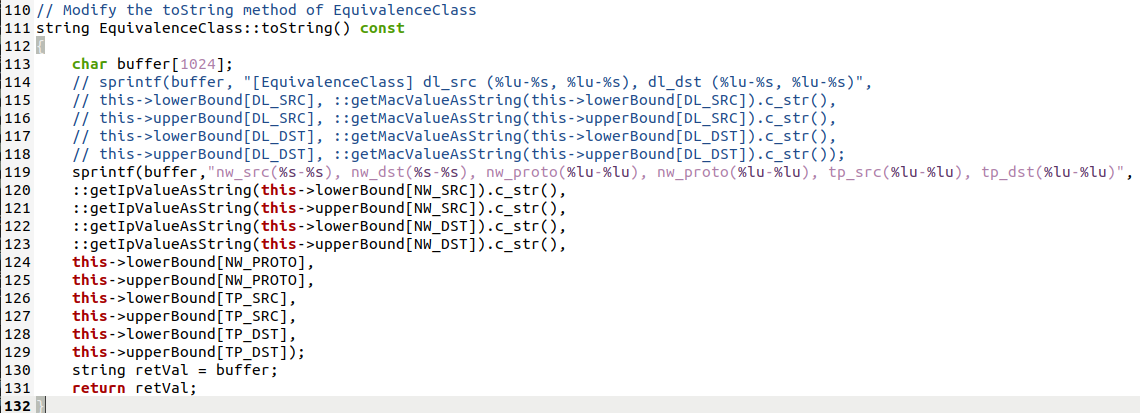


查看源码可以发现，打印EC信息的代码在VeriFlow.cpp文件的traverseForwardingGraph函数中，将传入的参数packetClass使用toString方法转换为字符串进行输出。

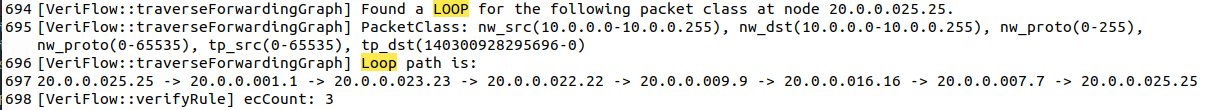


查看traverseForwardingGraph函数的定义我们可以知道，packetClass参数是EquivalenceClass类型，我们需要修改EquivalenceClass类内定义的toString方法。

EquivalenceClass类定义在EquivalenceClass.cpp文件中，按照实验指导书内的样例格式，我们修改类中的toString方法，修改后代码如下。



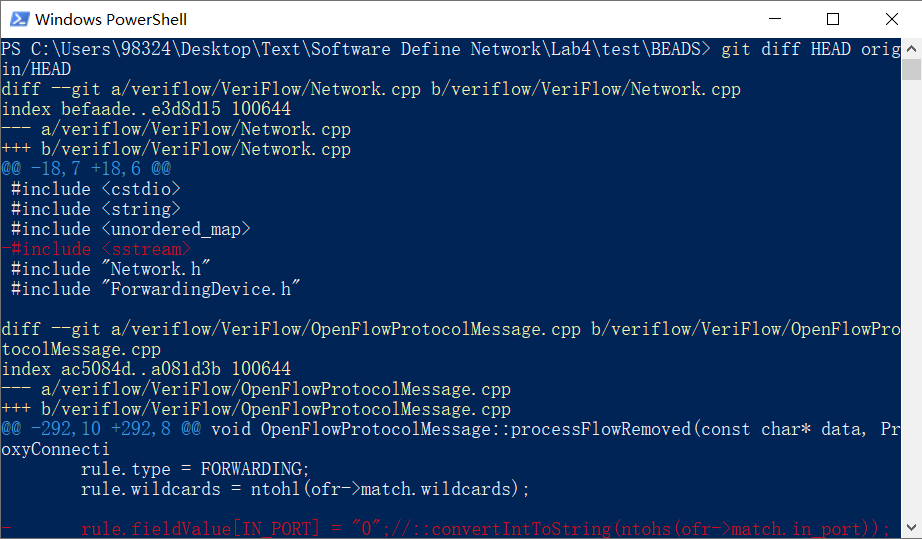
修改完成后重新进行VeriFlow的编译、启动Ryu控制器、启动VeriFlow的Proxy模式、启动拓扑、建立SDC与MIT之间的链接、下发路径、再次测试两主机连通性、查看log\_file.txt。



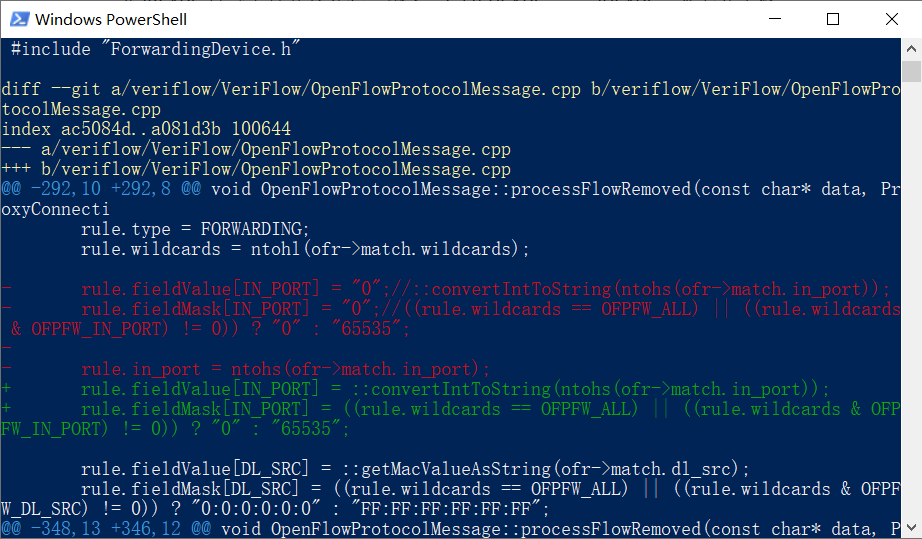
可以看到，log\_file.txt内打印了简化后的EC信息。

6. 分析原始代码与补丁代码的区别，思考为何需要添加补丁

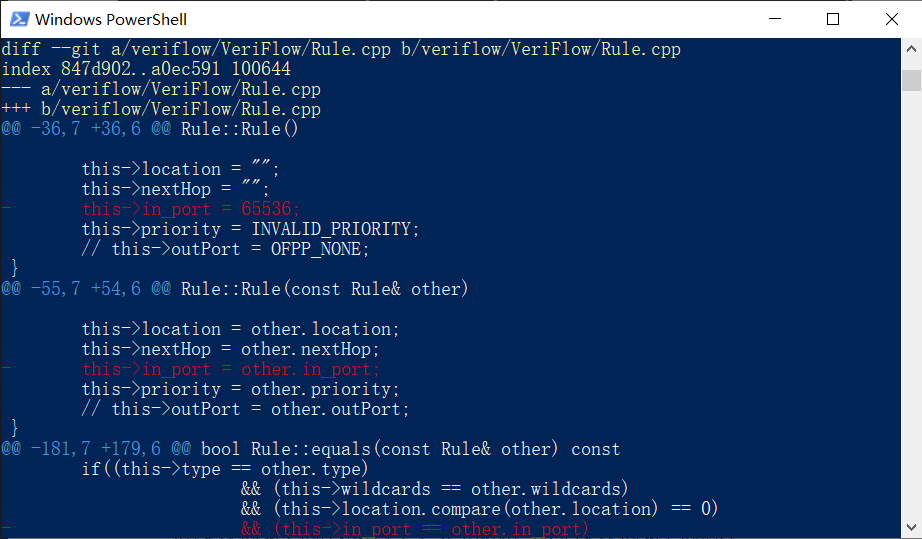
在/BEADS目录下打开命令行，输入git diff HEADS origin/HEADS，查看补丁修改的文件内容。



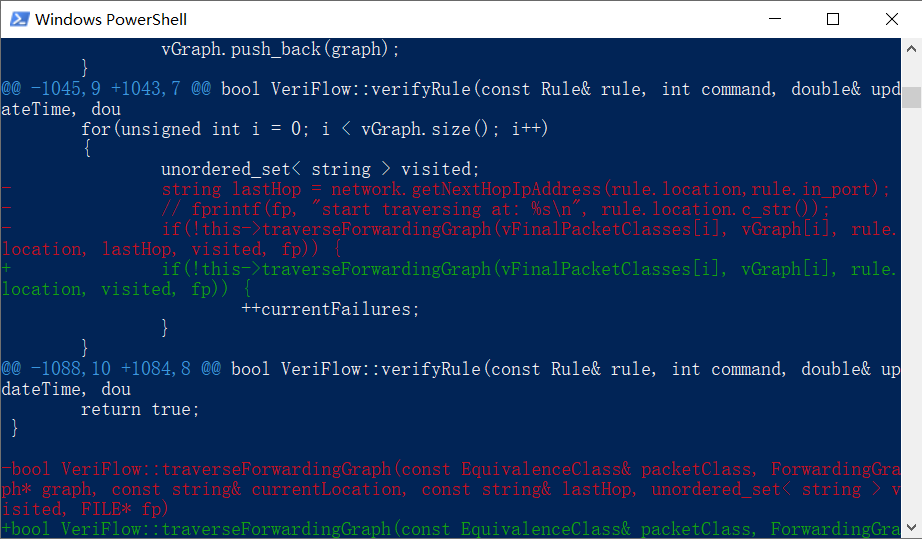
红色的内容是打上补丁后增加的内容，即原始文件相比于打上补丁的文件后减少的内容。其中，比较重要的修改内容如下。



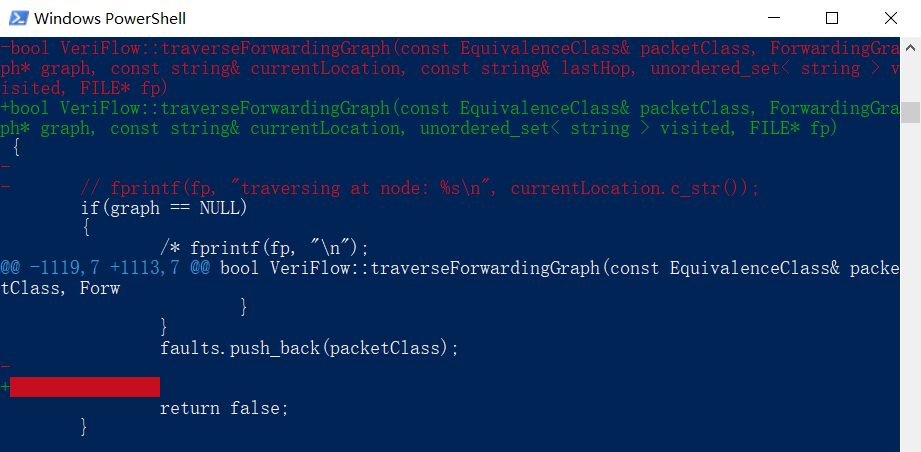
在OpenFlowProtocolMessage.cpp中增加了对rule对象中in\_port成员的处理。



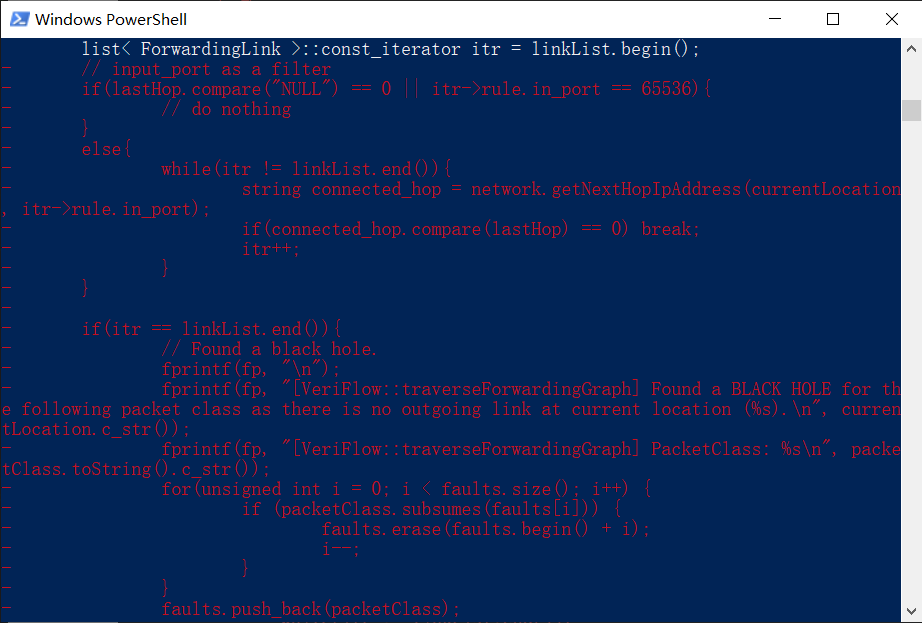
在Rule.cpp的Rule类中增加了in\_port成员，在多种构造函数中增加了对in\_port对象的初始化和存储。而原始文件没有区分in\_port，这样如果两个数据包的转发路径存在重叠的部分，会被认为是相同的数据通路，显然这样的方式划分的等价类和我们的目标是不一致的，会将等价类划分地更大，因此需要添加in\_port来保证正确地划分等价类。



在VeriFlow.cpp文件的verifyRule函数中增加了对lastHop的存储，提高了查找黑洞的能力。



在VeriFlow.cpp文件中修改了traverseForwardingGraph函数的定义，增加了lastHop参数，提高了查找黑洞的能力。



在VeriFlow.cpp文件的traverseForwardingGraph函数中增加了对黑洞的一种判断，完善了对黑洞的判断情况。

在github中开源代码中，给出了两种判断黑洞的情况：

1. 当前交换机或主机并不在当前网络中

2. 当前交换机或主机在网络中，但是无链路与其他交换机或主机相连

补丁中增加了一种判断黑洞的情况：

当前的交换机或者主机在网络的拓扑结构中，也存在与它相连的链路，但由于网络结构变化，使得从当前的交换机或者主机所在位置和相应端口(in\_port)，找不到上一跳的交换机或者主机。

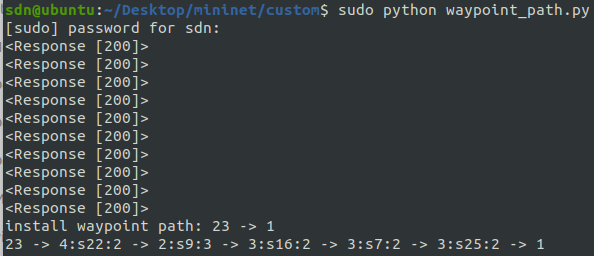
如果lastHop不是NULL且in\_port不是65536，则遍历当前结点的链接列表，根据当前交换机或主机的结点位置和对应端口查找上一跳(lastHop)的结点，如果遍历完成后仍找不到与lastHop对应的主机，则认为存在黑洞。

此外还有一些VeriFlow.h中关于traverseForwardingGraph函数声明的修改与调用traverseForwardingGraph函数时传入参数的修改。

准备知识：环路产生原因分析



在mininet命令行内输入SDC ping MIT后，可以在Ryu控制器里看到，我们建立了一条这样的路径，从10.0.0.18到10.0.0.12。

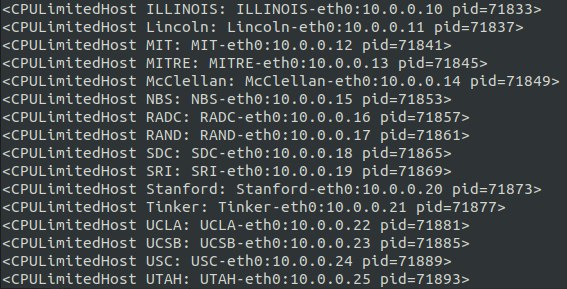


而下发路径后，我们可以看到，新增路径从交换机s23到交换机s1。新建立的链接路径也经过了交换机s22。交换机s23连接到交换机s22的端口4，再由交换机s22的端口2连接到交换机s9。

两条路径的重合结点为s22，s23，s1，s25。

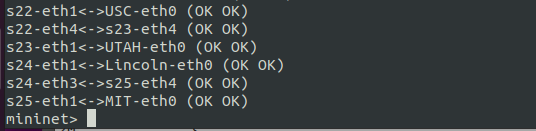
查看网络拓扑图可以看到，两条路径的重合结点是MIT主机所连接的交换机与USC主机所连接的交换机。

在mininet命令行内输入dump，查看各结点详细网络信息。



可以看到，MIT主机的IP地址为10.0.0.12，SDC主机的IP地址为10.0.0.18，USC主机的IP地址为10.0.0.24。MIT主机连接到交换机s15，交换机s15通过端口3连接到交换机s22的端口3，再由交换机s22的端口4连接到交换机s23的4端口。

更具体地，我们在mininet命令行内输入links，查看各结点之间连接的信息。



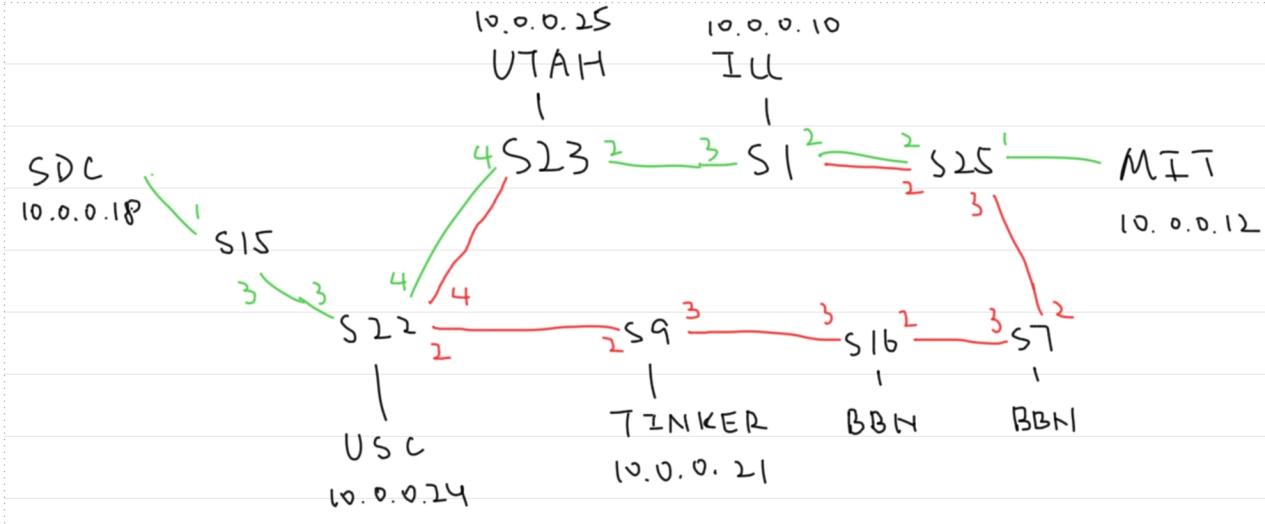
可以看到，MIT主机连接到交换机s24的端口1，USC主机连接到交换机s22的端口1。

原本的路径是交换机s1连接到交换机s25的端口2，再由交换机s25的端口1连接到MIT主机。

而下发路径后，交换机s1连接到交换机s25的端口2，再由交换机s25的端口3连接到交换机s7。

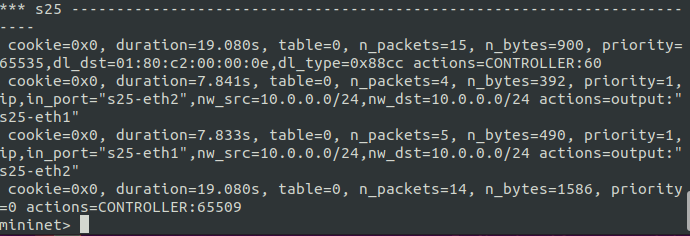
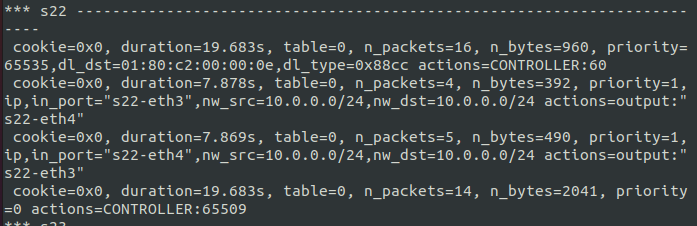
可以看到，我们应该分析的关键交换机是MIT主机所连接的交换机s25与USC主机所连接的交换机s22。

综合上述信息与分析，我们可以用如下的示意图来表示相关连接。



其中，绿色的路径代表原始路径，优先级值为1。红色的路径为下发路径，优先级值为10。

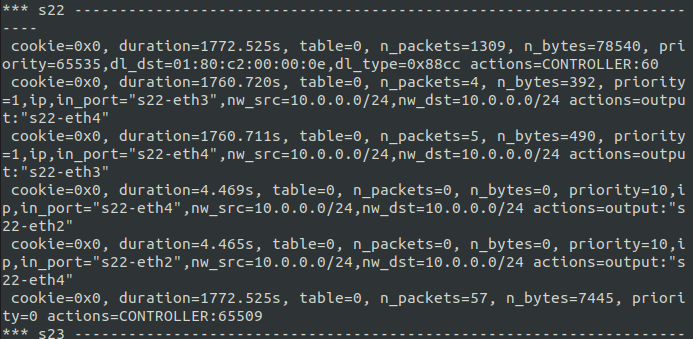
在未下发路径时，在mininet命令行内输入dpctl dump-flows，查看网络拓扑中各结点的流表。

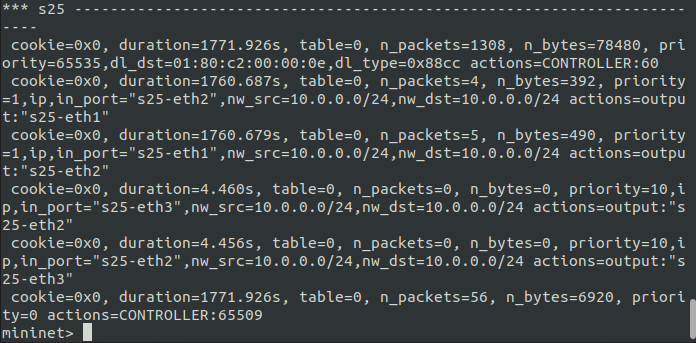


可以看到，交换机s22的流表中有4个表项，优先级的值分别为65535，1，1，0。其中in\_port为s22-eth3，output为s22-eth4的表项priority值为1。in\_port为s22-eth4，output为s22-eth3的表项priority值为1。

交换机s25的流表中有4个表项，优先级的值分别为65535，1，1，0。其中in\_port为s25-eth2，output为s25-eth1的表项priority值为1。in\_port为s25-eth1，output为s25-eth2的表项priority值为1。

下发路径后，在mininet命令行内输入dpctl dump-flows，查看网络拓扑中各结点的流表。





可以看到，交换机s22的流表中有6个表项，新增了2个表项，优先级的值分别为65535，1，1，10，10，0，增加了2条优先级的值为10的流表项。而其中in\_port为s22-eth4，output为s22-eth2的表项priority值为10。in\_port为s22-eth2，output为s22-eth4的表项priority值为10。

交换机s25的流表中有6个表项，新增了2个表项，优先级的值分别为65535，1，1，10，10，0，增加了2条优先级的值为10的流表项。其中in\_port为s25-eth3，output为s25-eth2的表项priority值为10。in\_port为s25-eth2，output为s25-eth3的表项priority值为10。

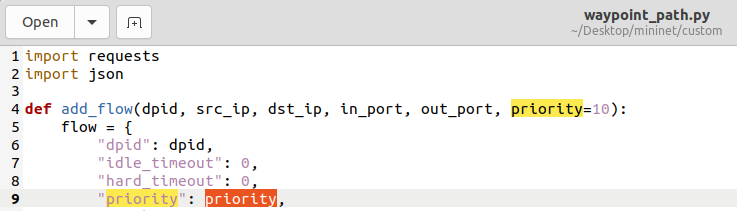
综合交换机s22与s25的流表与路径进行分析：

SDC主机向MIT主机发送ping request，数据包由端口3送到交换机s22，根据流表由端口4送到交换机s23，再转发给s1，s1转发给交换机s25的端口2。而交换机s25中有两条in\_port为端口2的流表项，其中一条为原始路径，通过端口1转发给MIT主机，优先级为1。另外一条为下发路径，通过端口3转发给s7，优先级为10。由于下发路径的流表项优先级更高，会优先匹配这条流表项，所以ping request数据包会往s7进行转发，而不会发给MIT主机，所以无法ping通。

而交换机s7转发给s16，s16转发给s9，s9转发给s22，交换机s22的流表项中，in\_port为2的流表项转发给端口4，即s22又转发给s23，这样就形成了一条由原始路径与下发路径共同组成的环路。

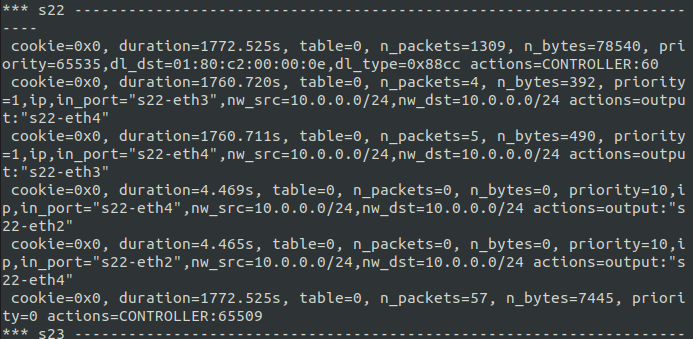
7. 拓展实验：修改waypoint\_path.py中的优先级字段。

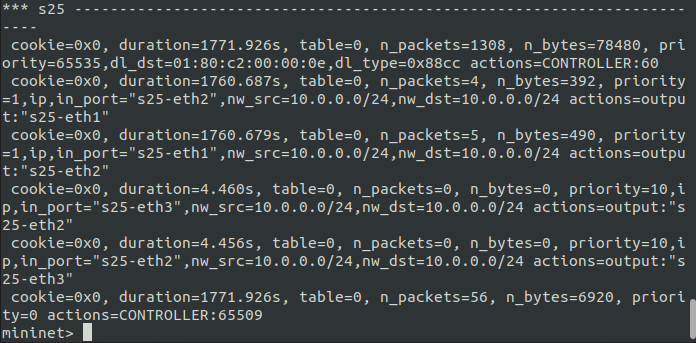
修改 waypoint\_path.py 代码中被添加规则的优先级字段，VeriFlow的检测结果会出错。



在waypoint\_path.py文件的add\_flow函数中，如果没有传入指定的priority参数，此参数会被默认赋值为10。

在未修改priority默认值时，在mininet命令行内输入dpctl dump-flows，查看网络拓扑中各结点的流表。

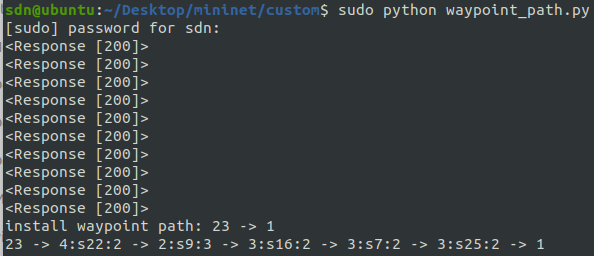




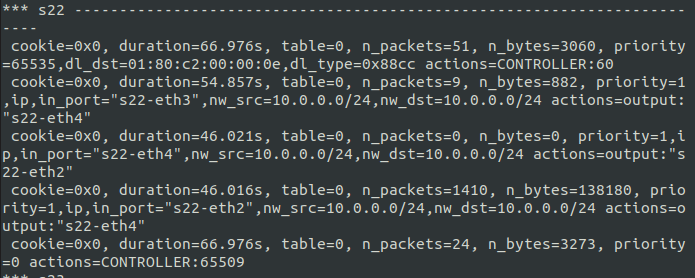
通过之前的分析我们可以看到，形成环路的关键结点是交换机s22与s25。我们还是对这两个交换机流表项进行分析，交换机s22的流表中有6个表项，优先级的值分别为65535，1，1，10，10，0。其中in\_port为s22-eth4，output为s22-eth2的表项priority值为10。in\_port为s22-eth2，output为s22-eth4的表项priority值为10。

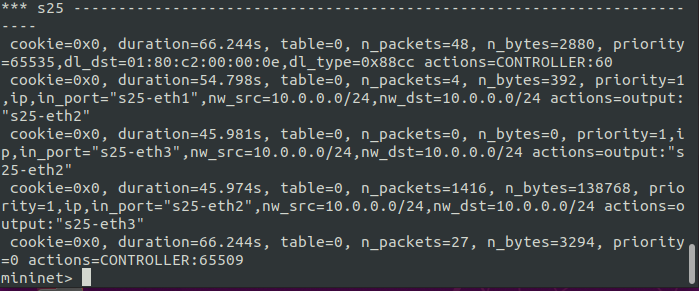
交换机s25的流表中有6个表项，优先级的值分别为65535，1，1，10，10，0。其中in\_port为s25-eth3，output为s25-eth2的表项priority值为10。in\_port为s25-eth2，output为s25-eth3的表项priority值为10。

修改waypoint\_path.py中的priority默认值后，重新启动Ryu控制器、启动VeriFlow的Proxy模式、启动拓扑、建立SDC与MIT之间的链接、下发路径、再次测试两主机连通性、查看log\_file.txt。



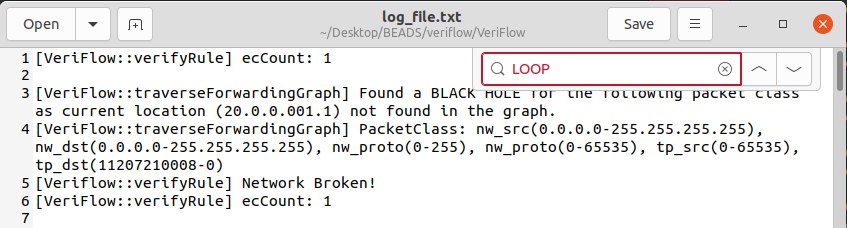
在mininet命令行内输入dpctl dump-flows，查看网络拓扑中各结点的流表。





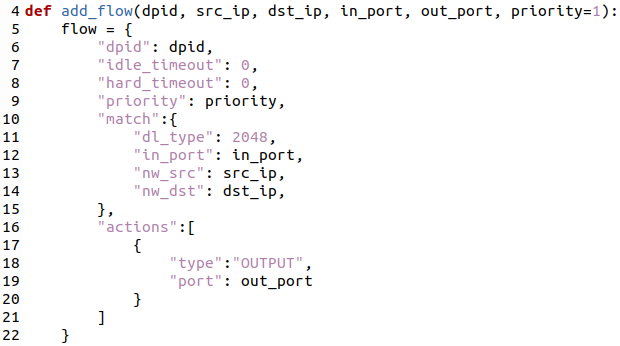
可以看到，交换机s22的流表中只有5个流表项，优先级的值分别为65535，1，1，1，0。仔细与未下发路径时的流表和未修改priority值下发路径后的流表相比较可以发现，缺少了由端口4发往端口3的流表项。

交换机s25的流表中只有5个流表项，优先级的值分别为65535，1，1，1，0。仔细与未下发路径时的流表和未修改priority值下发路径后的流表相比较可以发现，缺少了由端口2发往端口1的流表项。



可以看到，SDC ping MIT因为存在环路，依旧无法ping通，但是此时的log\_file.txt文件内没有环路路径LOOP的记录，VeriFlow没有检测到环路。

查看waypoint\_path.py文件中对增加流表项时匹配域的定义。



可以看到，增加流表项时，匹配域(match)金由dl\_type, in\_port, nw\_src, nw\_dst这4项，而out\_port属于actions，不属于匹配域，不参与匹配。

以交换机s22为例，而未下发路径时，s22流表中存在由端口4发往端口3的流表项，in\_port为4，priority为1。而未修改priority的值时，下发路径后，增加了由端口4发往端口2的流表项，in\_port也为4，但priority为10，因此可以与上述这条流表项区分开，不会覆盖这条流表项，而是新增了一条流表项。

但是修改priority值为1，下发路径后，新增的流表项由端口4发往端口2，in\_port也为4，priority也为1，虽然out\_port不同，但是匹配域的in\_port相同，因此覆盖了旧的流表项，而不是新增一条流表项。交换机s25的分析也同理，不再赘述。

即当匹配域相同时，新流表项覆盖了旧的流表项，事实上存在环路，但VeriFlow并没有检测出来环路。

而无法发现环路原因是判断环路选择下一跳时，VeriFlow利用priority字段进行了排序，当出现priority相同的规则时，就会出现问题。

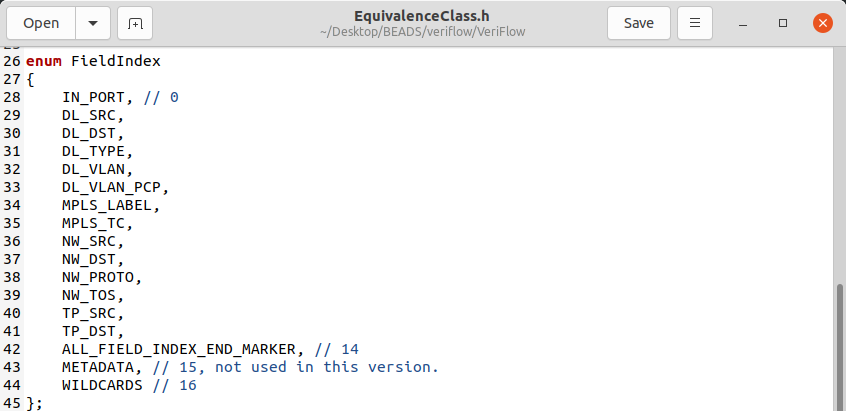
调用了graph->links[currentLocation].sort(compareForwardingLink);

当出现规则完全匹配需要对原来的规则进行覆盖的时候，VeriFlow并没有将原来的规则删除并加上新的规则，而是保留了原来的规则并抛弃了新加的规则。这就导致了VeriFlow无法检测到因为下发路径而生成的环路。

8. 拓展实验：域的验证

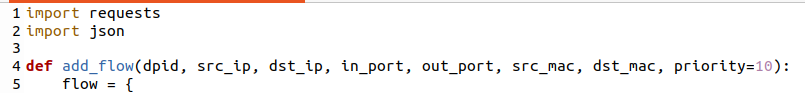
在VeriFlow支持的14个域中，挑选多个域（不少于5个）进行验证，输出并分析结果。

在EquivalenceClass.h文件中，我们可以看到VeriFlow支持的域。

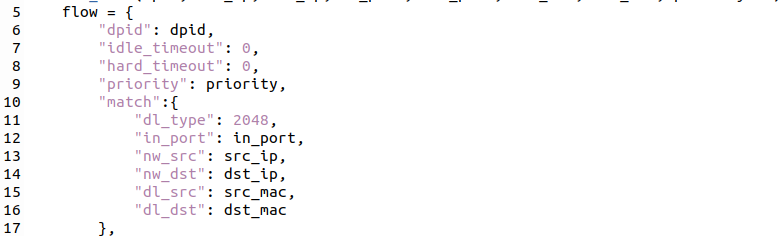


结合原有的waypoint\_path.py文件，我们选择in\_port, dl\_src, dl\_dst, dl\_type, nw\_src, nw\_dst这6个域进行验证。

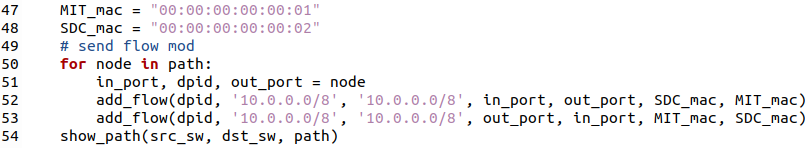
恢复基础实验中对EC信息打印格式的修改，然后修改waypoint\_path.py如下：



修改了add\_flow函数的定义，添加了两个传入参数src\_mac与dst\_mac。

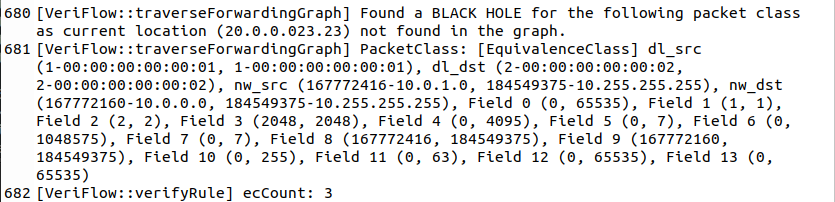


在匹配域中增加了dl\_src与dl\_dst两个匹配域。

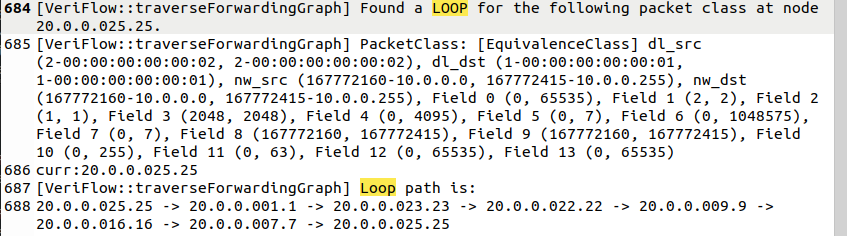


在install\_path函数中增加了MIT与SDC的mac地址，在添加流表项的时候将这两个mac地址作为参数传入。其余部分保存waypoint\_path.py原有代码，不做修改。

修改waypoint\_path.py文件后，重新启动Ryu控制器、启动VeriFlow的Proxy模式、启动拓扑、建立SDC与MIT之间的链接、下发路径、再次测试两主机连通性、查看log\_file.txt。



可以看到，受影响的等价类(EC)个数为3，即Field 1，2和3一共3个等价类，与基础实验部分相同。dl\_src为MIT\_mac，dl\_dst为SDC\_mac。



可以看到，环路路径LOOP中，nw\_src与nw\_dst相同，代表存在环路，且打印出了正确的环路路径：25 -> 1 -> 23 -> 22 -> 9 -> 16 -> 7 -> 25，与设定的环路路径相同。

**实验结果：**

1. 学习了如何通过下发路径形成转发环路。

2. 学习了如何使用VeriFlow检测黑洞和环路。

3. 熟悉了VeriFlow的部分源代码和数据结构，自己动手修改源代码完成实验要求。

4. 更加深入地了解了等价类的概念，学习了等价类中包含的信息。

5. 学习了流表中优先级的概念，与添加流表项时可能产生的覆盖。

**源代码：**

waypoint\_path.py

import requests

import json

def add\_flow(dpid, src\_ip, dst\_ip, in\_port, out\_port, src\_mac, dst\_mac, priority=10):

    flow = {

        "dpid": dpid,

        "idle\_timeout": 0,

        "hard\_timeout": 0,

        "priority": priority,

        "match":{

            "dl\_type": 2048,

            "in\_port": in\_port,

            "nw\_src": src\_ip,

            "nw\_dst": dst\_ip,

            "dl\_src": src\_mac,

            "dl\_dst": dst\_mac

        },

        "actions":[

            {

                "type":"OUTPUT",

                "port": out\_port

            }

        ]

    }

    url = 'http://localhost:8080/stats/flowentry/add'

    ret = requests.post(

        url, headers={'Accept': 'application/json', 'Accept': 'application/json'}, data=json.dumps(flow))

    print(ret)

def show\_path(src, dst, port\_path):

    print('install waypoint path: {} -> {}'.format(src, dst))

    path = str(src) + ' -> '

    for node in port\_path:

        path += '{}:s{}:{}'.format(\*node) + ' -> '

    path += str(dst)

    path += '\n'

    print(path)

def install\_path():

    '23 -> 4:s22:2 -> 2:s9:3 -> 3:s16:2 -> 3:s7:2 -> 3:25:2 -> 1'

    src\_sw, dst\_sw = 23, 1

    waypoint\_sw = 9  # Tinker 10.0.0.21, s9

    path = [(4, 22, 2), (2, 9, 3), (3, 16, 2), (3, 7, 2), (3, 25, 2)]

    # path = [(3, 7 , 2)]

    MIT\_mac = "00:00:00:00:00:01"

    SDC\_mac = "00:00:00:00:00:02"

    # send flow mod

    for node in path:

        in\_port, dpid, out\_port = node

        add\_flow(dpid, '10.0.0.0/8', '10.0.0.0/8', in\_port, out\_port, SDC\_mac, MIT\_mac)

        add\_flow(dpid, '10.0.0.0/8', '10.0.0.0/8', out\_port, in\_port, MIT\_mac, SDC\_mac)

    show\_path(src\_sw, dst\_sw, path)

if \_\_name\_\_ == '\_\_main\_\_':

install\_path()

VeriFlow.cpp

bool VeriFlow::verifyRule(const Rule& rule, int command, double& updateTime, double& packetClassSearchTime, double& graphBuildTime, double& queryTime, unsigned long& ecCount, FILE\* fp)

{

    // fprintf(fp, "[VeriFlow::verifyRule] verifying this rule: %s\n", rule.toString().c\_str());

    ......

    if(ecCount == 0)

    {

        fprintf(stderr, "[VeriFlow::verifyRule] Error in rule: %s\n", rule.toString().c\_str());

        fprintf(stderr, "[VeriFlow::verifyRule] Error: (ecCount = vFinalPacketClasses.size() = 0). Terminating process.\n");

        exit(1);

    }

    else

    {

        fprintf(stdout, "\n");

        fprintf(stdout, "[VeriFlow::verifyRule] ecCount: %lu\n", ecCount); // output to the veriflow console

        fprintf(fp, "[VeriFlow::verifyRule] ecCount: %lu\n", ecCount); // output to the FILE \*fp, which means the log\_file.txt

}

// check if currentLocation is in path

static bool isInPath(const vector<string>& path, const string& location)

{

    int n = path.size(); //// get the number of nodes on loop

    for(int i =0 ;i < n;i++) // go through the path

    {

        if(path[i] == location)

        {

            return true;

        }

    }

    return false;

}

// print loop and infomation

static void printLoop(FILE\* fp, const vector<string>& path, const string& current)

{

    fprintf(fp, "[VeriFlow::traverseForwardingGraph] Loop path is:\n");

    int n = path.size(); // get the number of nodes on loop

    for(int i = 0;i < n;i++)

    {

        fprintf(fp, "%s -> ", path[i].c\_str()); // print path nodes

    }

    fprintf(fp, "%s\n", current.c\_str()); // print current node

}

// modify the arguments

bool VeriFlow::traverseForwardingGraph(const EquivalenceClass& packetClass, ForwardingGraph\* graph, const string& currentLocation, const string& lastHop, unordered\_set< string > visited, vector<string>& path, FILE\* fp)

{

......

if(visited.find(currentLocation) != visited.end())

    {

        // Found a loop.

        fprintf(fp, "\n");

        fprintf(fp, "[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node %s.\n", currentLocation.c\_str());

        fprintf(fp, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n", packetClass.toString().c\_str());

        //printloop

        fprintf(fp, "curr:%s\n", currentLocation.c\_str());

        printLoop(fp, path, currentLocation);

        for(unsigned int i = 0; i < faults.size(); i++) {

            if (packetClass.subsumes(faults[i])) {

                faults.erase(faults.begin() + i);

                i--;

            }

        }

        faults.push\_back(packetClass);

        return false;

    }

    visited.insert(currentLocation);

    if(!isInPath(path, currentLocation))

    {

        path.push\_back(currentLocation); // If not, add currentLocation to path

    }

EquivalenceClass.cpp

// Modify the toString method of EquivalenceClass

string EquivalenceClass::toString() const

{

    char buffer[1024];

    sprintf(buffer,"nw\_src(%s-%s), nw\_dst(%s-%s), nw\_proto(%lu-%lu), nw\_proto(%lu-%lu), tp\_src(%lu-%lu), tp\_dst(%lu-%lu)",

    ::getIpValueAsString(this->lowerBound[NW\_SRC]).c\_str(),

    ::getIpValueAsString(this->upperBound[NW\_SRC]).c\_str(),

    ::getIpValueAsString(this->lowerBound[NW\_DST]).c\_str(),

    ::getIpValueAsString(this->upperBound[NW\_DST]).c\_str(),

    this->lowerBound[NW\_PROTO],

    this->upperBound[NW\_PROTO],

    this->lowerBound[TP\_SRC],

    this->upperBound[TP\_SRC],

    this->lowerBound[TP\_DST],

    this->upperBound[TP\_DST]);

    string retVal = buffer;

    return retVal;

}