# 实验目的

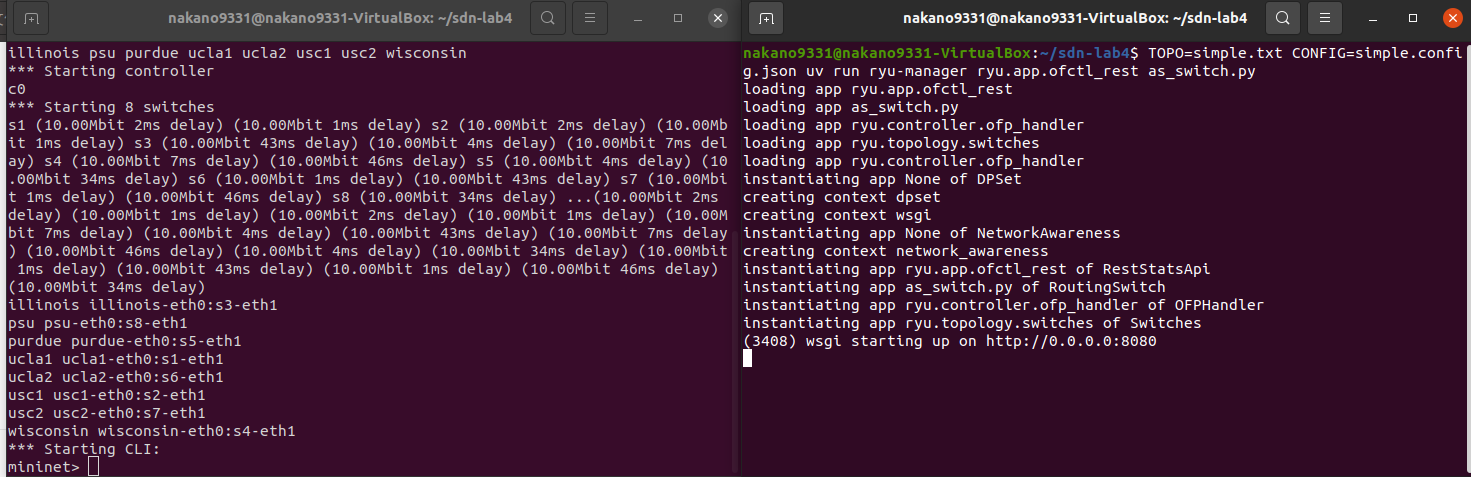
1. 理解网络中网络故障出现的必然性
2. 理解网络验证工具VeriFlow的原理
3. 掌握VeriFlow的检测网络故障的方法
4. 提高阅读工程代码、修改代码的能力

# 实验过程

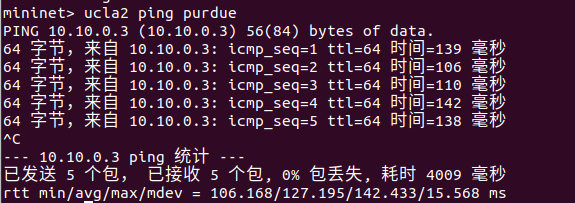
## 2.1 热身实验部分

### 2.1.1 观察转发环路

步骤一：启动拓扑和控制程序。



步骤二：在拓扑中ucla2 ping purdue建立连接。



此时观察通讯路径，控制器中显示为6 -> s6:3 -> s3:3 -> s5:1 -> 10.10.0.3。



步骤三：执行uv run ./gen\_loop.py下发新路径。

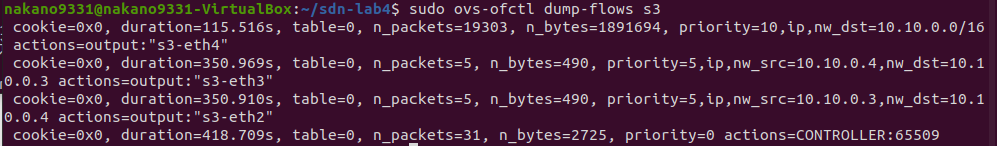


执行gen\_loop.py下发从illinois途经wisconsin到达ucla2的路径之后，尝试ucla2 ping purdue失败，具体情况如下所示。

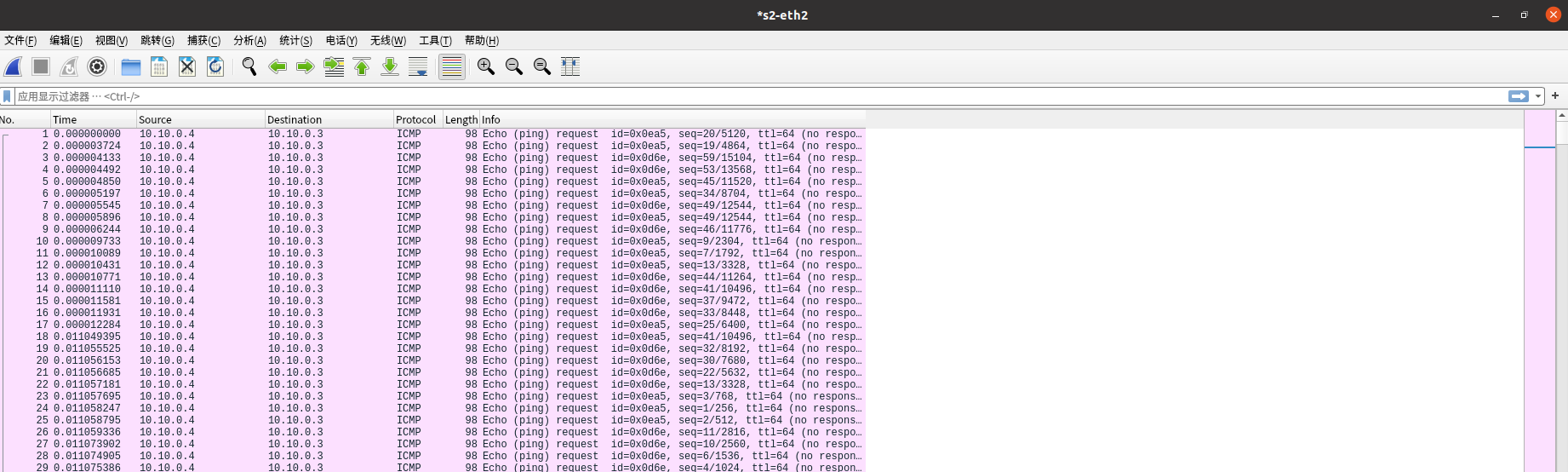


步骤四：查看路径上某一个交换机，如illinois的流表。

执行sudo ovs-ofctl dump-flows s3发现匹配某一条流表的数据包数目异常增加，高达19303个packet，具体情况如下所示，说明了路由环路的存在。

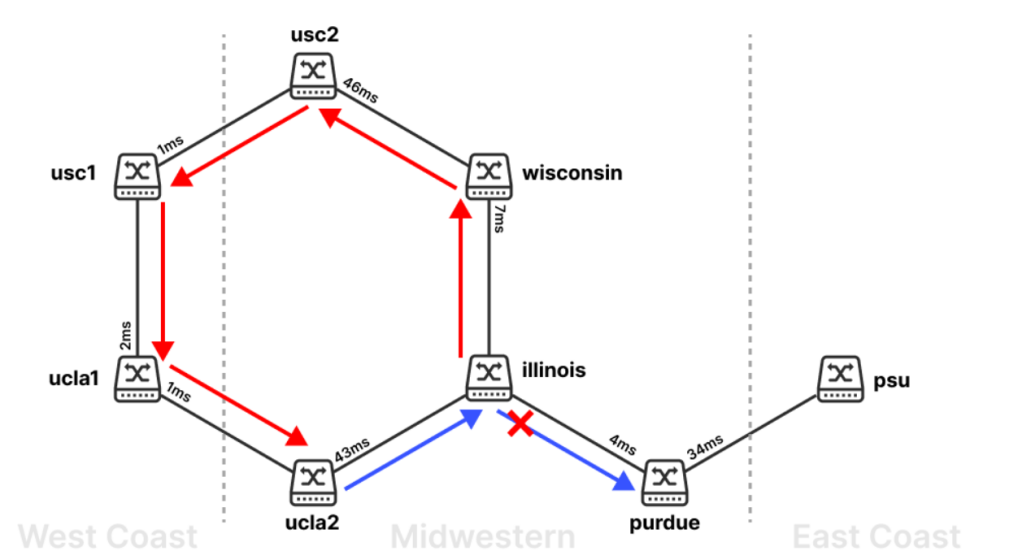


使用Wireshark观察S2-eth2端口，发现大量源IP地址为10.0.0.4，目的IP地址为10.0.0.3的ICMP请求报文，这也从另一方面说明了路由环路的存在。



路径的重新下发导致了形如S3处的流表改变，导致 ILLINOIS 到 PURDUE 的数据包转发错误，即controller控制全局Switch的FlowTable，最开声明使S3

数据包发往S5，随后加⼊新链路又使得S3发往S4，故形成了环路。



S4

S5

S3

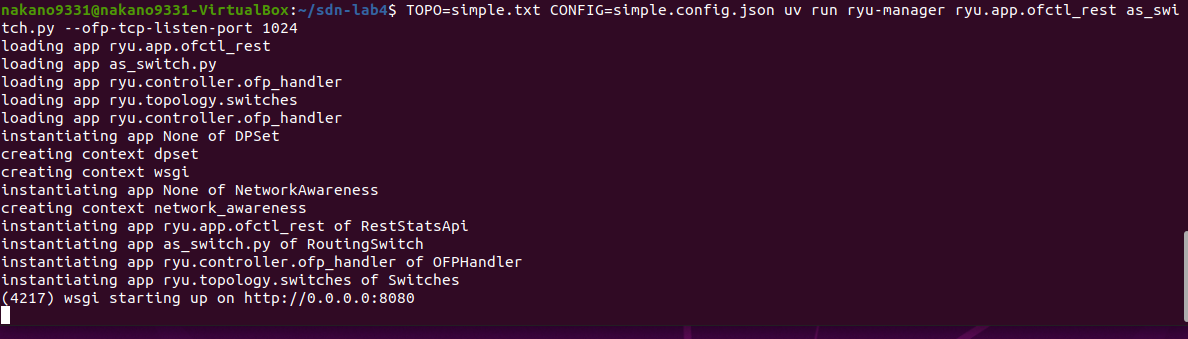
S3

### 2.1.2 使用VeriFlow验证环路问题

步骤一：编译VeriFlow.

执行命令make -C veriflow -j$(nproc)进行编译，其中，-j$(nproc)为指定编译线程数，可以不加。

步骤二：在自定义端口开启远程控制器，运行AS控制器程序，执行命令后结果如下所示。

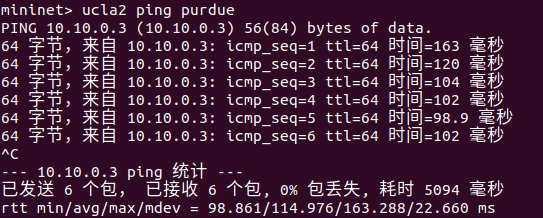


步骤三：执行./VeriFlow 6633 127.0.0.1 1024 simple.txt veriflow.log命令运行VeriFlow的proxy模式。

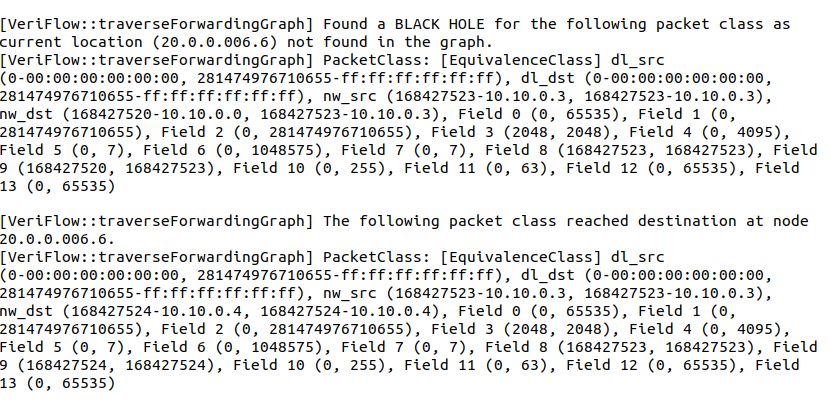
步骤四：启动拓扑，并在拓扑中ucla2 ping purdue建立连接。

步骤五：再次执行uv run ./gen\_loop.py下发从illinois途经wisconsin到达ucla2的路径，在log文件中观察VeriFlow检测到的环路信息。

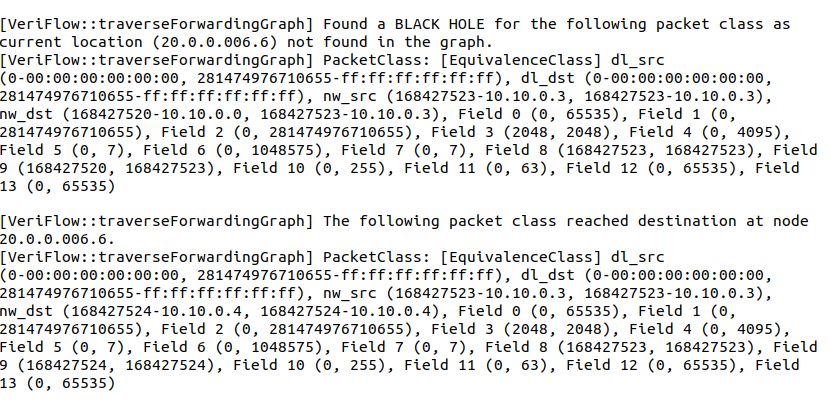
首先观察到，在ucla2 ping purdue建立连接后，生成转发环路的过程与先前操作过程相同。在下发从UTAH途经TINKER到达ILLINOIS的路径后，log文件中VeriFlow记录的信息如下所示。



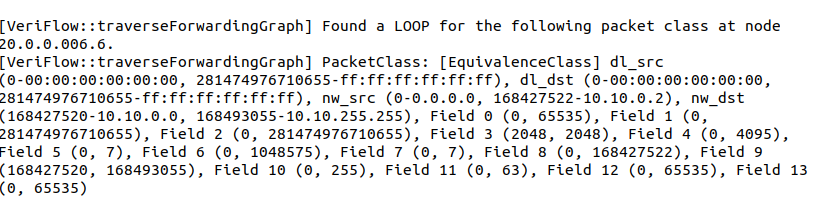
由于控制器的工作特性限制，其无法在同一时间内同时下发多个流表项。在这种情况下，VeriFlow对特定流表项进行检查时，可能会出现以下情况。



当数据包依据当前流表项匹配规则被转发至下一个交换机后，却发现下一个交换机中尚未部署用于处理该数据包的相应流表项。面对这种状况，VeriFlow会对黑洞信息进行记录。



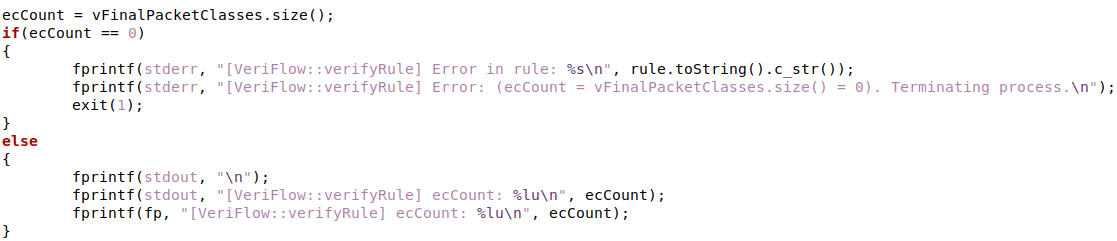
当某条流表项具备使数据包顺利转发至目的主机的能力时，这意味着目的主机可以通过该流表项实现可达。此时，VeriFlow会将这一目的主机可达信息进行记录，为网络拓扑和流量管理提供参考依据。



在新流表项的下发过程中，VeriFlow会对新流表项可能对网络路由产生的影响进行实时监测。一旦发现新流表项的引入会导致路由环路的产生，VeriFlow在日志文件中记录该路由环路信息。

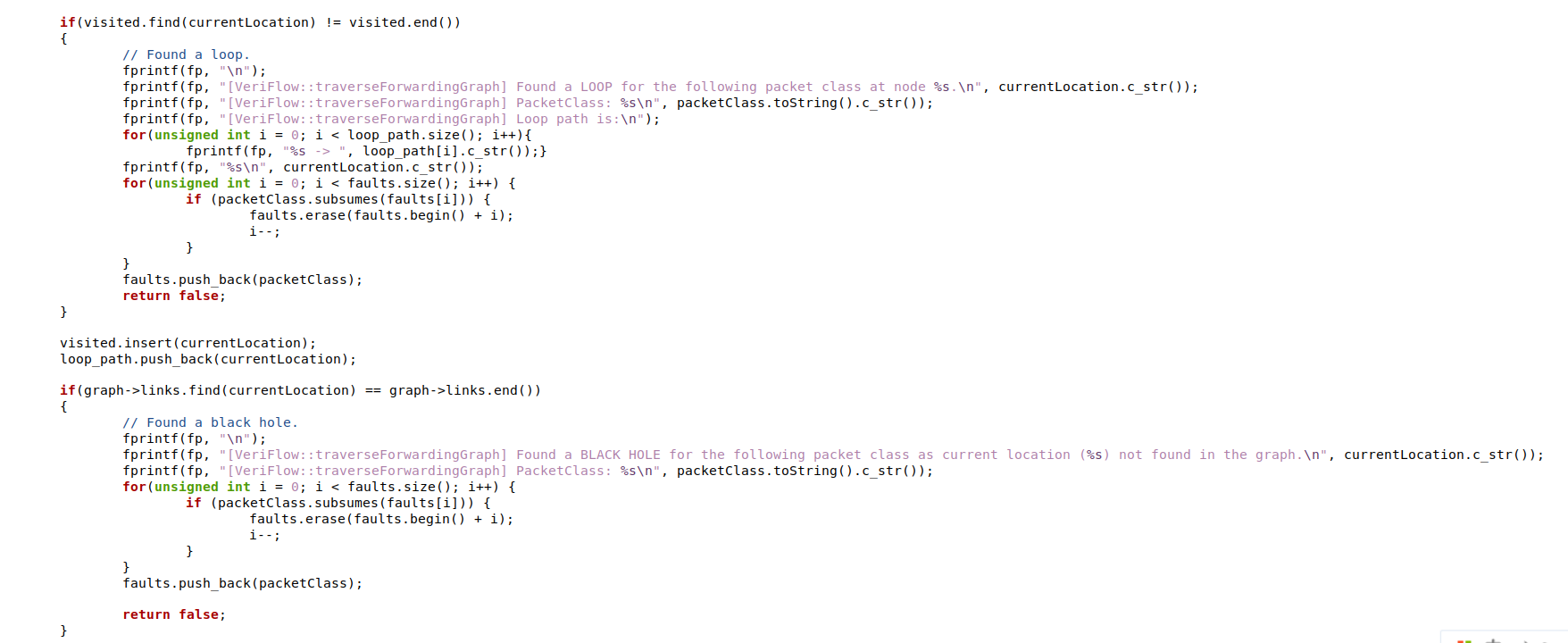
### 2.1.3 VeriFlow 内容修改

步骤一：输出每次影响EC的数量。VeriFlow::verifyRule()为执行 VeriFlow 核心算法的函数，包括对等价类的划分、转发图的构造与不变量的验证。函数中变量 ecCount 为 EC 数目，将等价类定向输出到 fp，即 log\_file.txt ⽂件中。



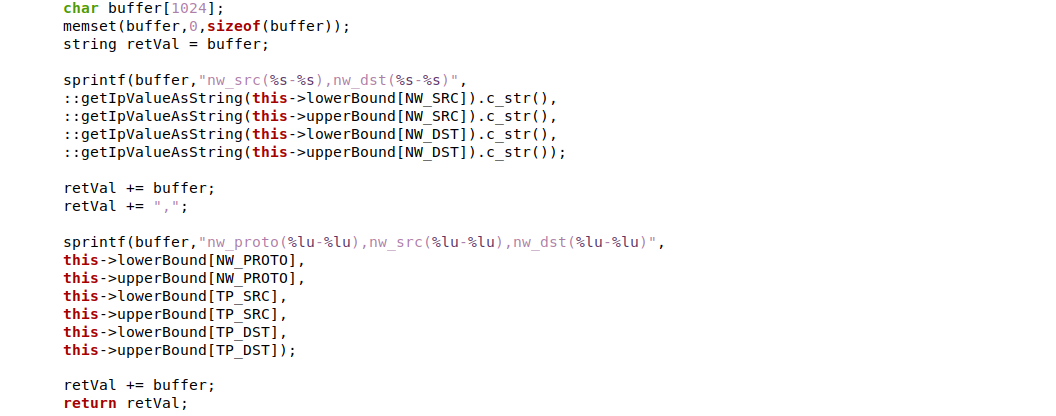
步骤二：输出环路路径详细信息。在 VeriFlow::traverseForwardingGraph() 函数中，系统将对特定等价类的转发图展开遍历操作，以此验证网络中是否存在环路或黑洞问题。

在该函数执行过程中，visited 变量被用于记录已经遍历过的节点。当检测到当前节点已存在于 visited 记录中意味着环路产生。利用哈希技术实现元素的无序存储，这种方式能够在保证数据存储高效性的同时，快速完成元素的查找判断。此处选择新增一个 vector<string> 类型的变量 loop\_path ，在遍历过程中实时记录路径信息，从而完整保存环路的具体走向。

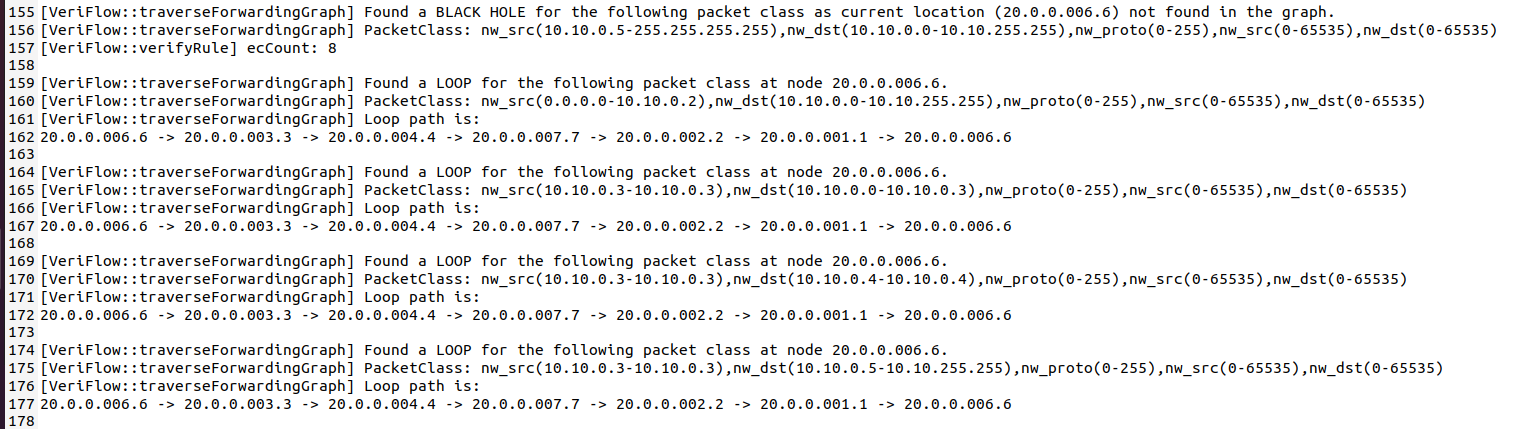


步骤三：进一步打印出环路对应的EC的相关信息。

在 EquivalenceClass 类中修改函数 toString() 函数，将原来的输出，改成输出标准 TCP/IP 五元组，即为(IP 源地址，IP ⽬的地址，⽹络层协议类型，传输层源端⼝号，传输层⽬的端⼝号)。



重复上一部分各个步骤，在log文件中观察VeriFlow检测到的环路信息，最终结果如下所示。



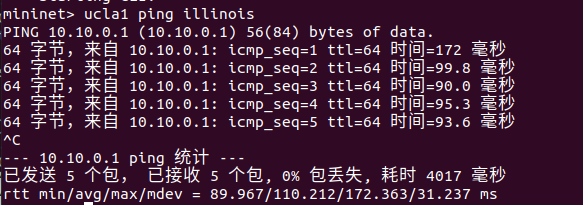
## 2.2 基础实验部分

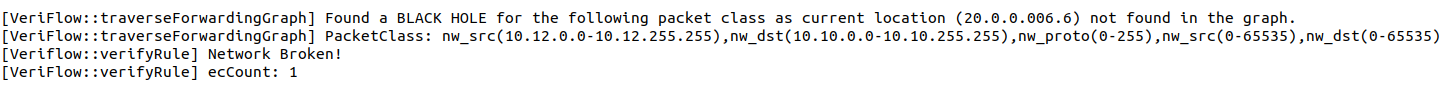
### 2.1.1 建立转发黑洞

步骤一：在自定义端口开启远程控制器，运行AS控制器程序。

步骤二：运行VeriFlow的proxy模式并启动拓扑。

步骤三：建立转发路径，观察到此时，两者可以正常ping通，但是VeriFlow提示出现了黑洞。



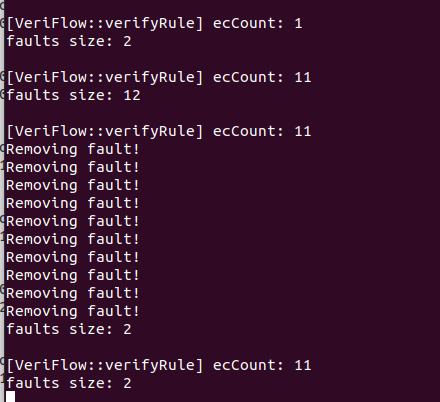




通过观察log记录，发现节点编号20.0.0.006.6，即S6交换机未正确建立转发路径，在该交换机上，找不到对应的转发条目来处理指定的包。ucla1发包到illinois时，走到ucla2处，不知道下一跳方向。而illinois发包回ucla1时，走到ucla2，也不知道怎么转发回去了。

在 fix\_path.py 中，通过 send\_flow\_mod 函数添加了流表项。如向S6下发规则，当接收到目的IP为 10.10.0.0/16 的数据包时，将其从端口3转发出去。对于S3和 S1则转发到端口1对应的主机。通过这一系列流表项的添加，为之前等价类的数据包在网络中指定了明确的转发路径。

然而，执行后 VeriFlow的发现的网络错误依旧存在。



### **2.1.2 修复Fault计算**

为了处理上述错误计算，修改VeriFlow.cpp中的traverseForwardingGraph函数，对转发图进行遍历，以此来检查数据包类在图中的转发状况，并且识别诸如环路、黑洞等问题。保留了原有对输入参数的检查、对已访问节点的检查、对当前节点是否存在出边的检查等关键步骤。对当前节点的出边列表以规则 compareForwardingLink 进行排序。根据输入端口过滤出边的逻辑，以防止数据包反向转发。

#### 2.1.2.1 VeriFlow.cpp修改部分

修改思路为调用 updateFaults 函数更新故障列表，该函数会对故障列表中的每个故障与当前数据包类求交集和差集，并且移除空的故障。此外，使用 isPacketClassInFaults 函数检查当前数据包类是否已存在于故障列表中。

具体部分见源代码处。

#### 2.1.2.2 EquivalenceClass.cpp修改部分

在EquivalenceClass.cpp中加入定义 EquivalenceClass 类的三个成员函数，分别是 intersection 、 difference 和 isEmpty 。

①Intersection 用于计算当前两个对象的交集。首先初始化新的上下界数组，长度为 ALL\_FIELD\_INDEX\_END\_MARKER 的数组并将其命名为 newLowerBound 和 newUpperBound ，用于存储交集的上下界。针对每个字段，取当前对象和 other 对象对应字段的下界的最大值作为新的下界，上界的最小值作为新的上界。 若所有字段的上下界都为 0，则认为交集为空。此时返回一个空对象；否则，使用新的上下界数组创建一个新的 EquivalenceClass 对象并返回。

②difference 用于计算当前两个对象的差集。首先初始化结果部分向量，创建一个 std::vector<EquivalenceClass> 类型的 resultParts ，用于存储差集的各个部分。对于每个字段，若 other 的上下界合法，则检查当前对象的上下界与 other 的上下界的关系，若存在差值部分，则创建新的 EquivalenceClass 对象并添加到 resultParts 中。 随后对 resultParts 中所有部分进行合并，通过多次遍历所有字段，将重叠部分合并成一个更大的等价类。

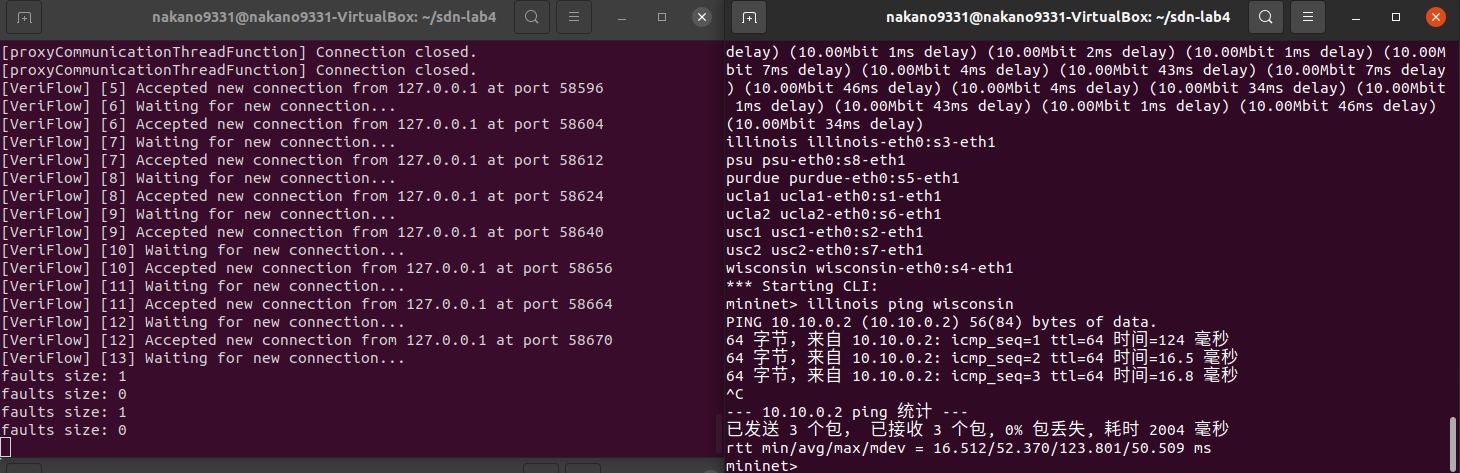
若 resultParts 不为空，将所有部分合并成一个最终的对象并返回；否则，返回一个空的 EquivalenceClass 对象。

③isEmpty 用于判断当前 EquivalenceClass 对象是否为空。遍历所有字段，若存在某个字段的上下界不为 0，则返回 false ；否则，返回 true 。

各个部分修改内容，详见源代码处。

#### 2.1.2.3 实验结果

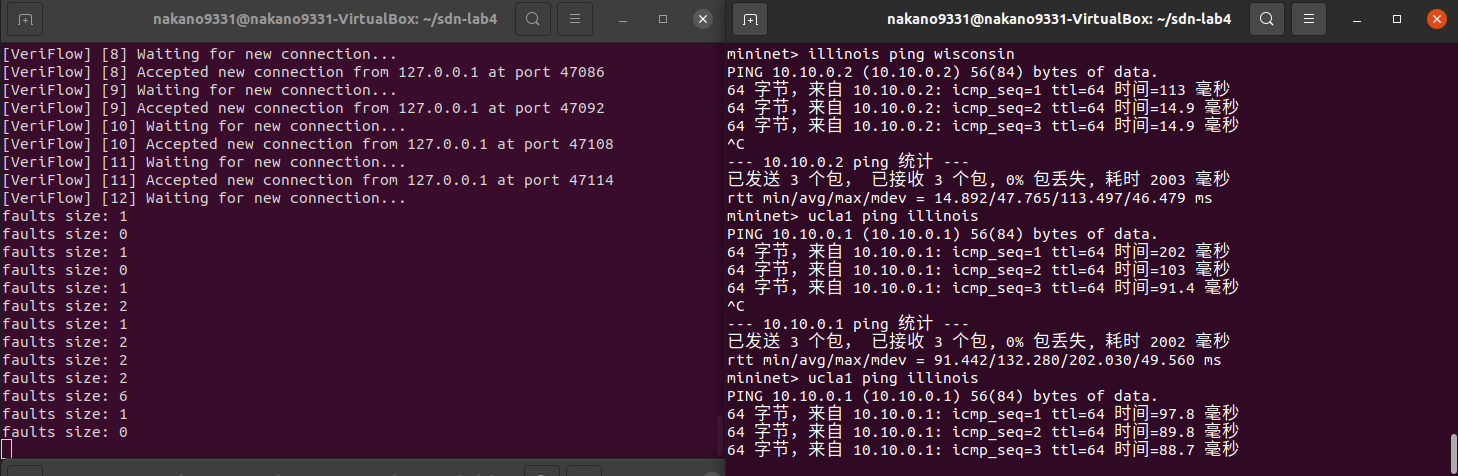
ucla1 ping illinois 之前，运⾏ illinois ping wisconsin 如下所示。



ucla1 ping illinois 之后，修复之前如下所示。



修复之后如下所示。



如图所示，通过处理新加入规则等价类和原有错误之间关系，成功将Faults计算问题修复。

## 2.3 拓展实验部分

本部分选择阅读 as\_switch.py 有关数据包转发的代码，解释产⽣⿊洞的原因。请说明，产⽣的⿊洞问题会不会影响⽹络的实际使⽤？修改 as\_switch.py ，使得同样的ping操作不会再产⽣⿊洞。

as\_switch.py 主要功能是处理网络中的数据包，进行二层自学习和三层路由转发。它首先获取数据包的相关信息，如数据路径、交换机 ID、输入端口 in\_port 等。然后解析数据包，获取以太网层、ARP 层和 IPv4 层的协议对象。接着进行二层自学习，将源 MAC 地址和输入端口的映射关系存储在 dpid\_mac\_port 中。最后根据数据包的类型，分别调用 handle\_arp 或 handle\_ipv4 方法进行处理。

handle\_arp 当目标 MAC 地址在 dpid\_mac\_port 中有对应的输出端口，则直接将数据包从该端口发送出去。否则，将数据包发送到所有除源主机所在的端口以外，连接主机的交换机端口。

handle\_ipv4 获取当前交换机的子网 switch\_net ，然后查找源 IP 地址所属的子网 srcnet 和目标 IP 地址所属的子网 dstnet 。如果目标 IP 地址在当前交换机的子网内，则进行本地交换， gateways 设为 None ；否则，查找目标 IP 地址所属子网的网关 gateways 。如果无法识别源或目标子网，则无法转发数据包并返回。

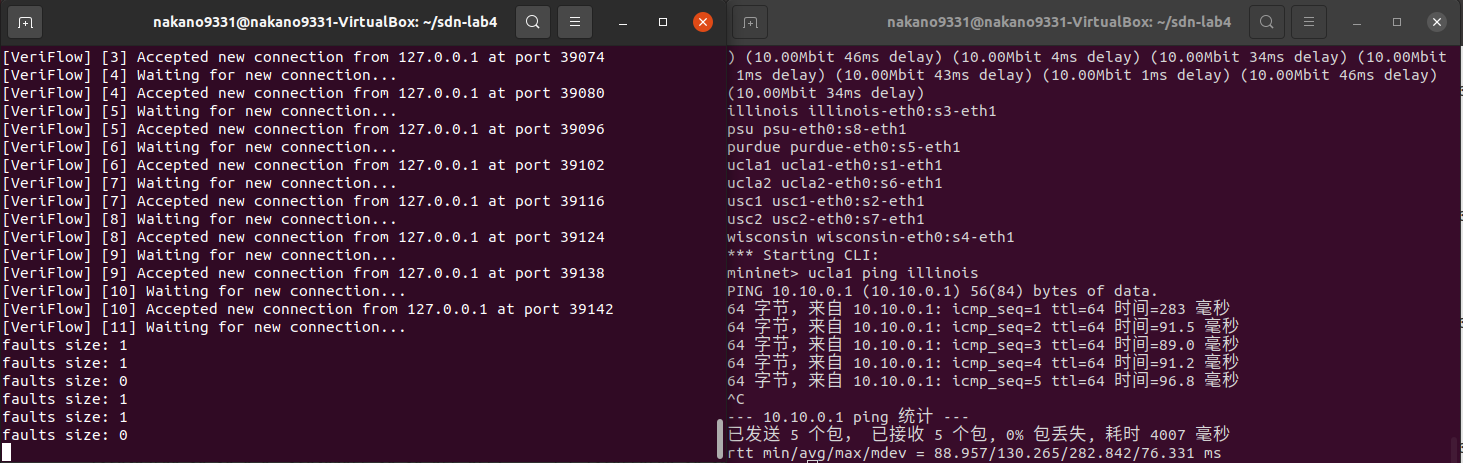
add\_path 是一个内部函数，用于添加从源到目标的路径。它首先根据路径 route 确定每个跳的输出端口，记录在 port\_path 中。然后展示路径信息，并在每个路径节点上发送流表项，最后返回当前交换机的输出端口。

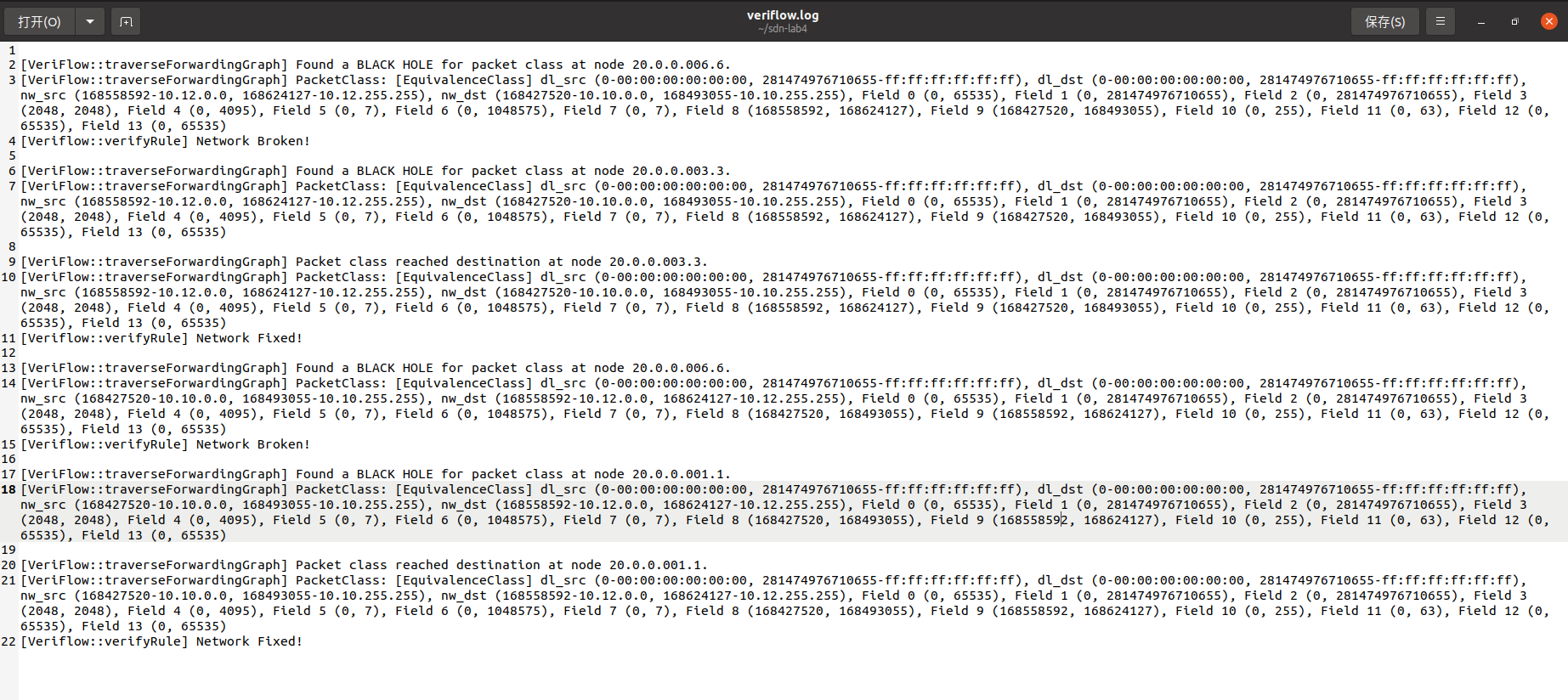
根据 gateways 是否为 None 来判断是进行 AS 内的数据包交换还是 AS 间的数据包交换。如果是 AS 内交换，获取从当前交换机到目标 IP 地址的最短路径 dpid\_path ，并调用 add\_path 添加路径；如果是 AS 间交换，判断当前交换机是否为网关，如果是则发送到对等网关，否则找到最近的网关，获取到该网关的最短路径并添加路径。在安装路径后，根据获取的输出端口 out\_port ，构造 OFPPacketOut 消息，将数据包从该端口发送出去。

产⽣⿊洞的原因在于跨 AS 后进入了 AS 内交换的逻辑中，开始计算从当前交换机到目标 IP 地址的最短路径 dpid\_path ，并调用 add\_path 添加路径，是以准确的IP为目标下发流表，当遇到以 /16 的子网目标时，查询不到相应流表导致出现黑洞，这种黑洞错误若是在以相同的源地址和目标地址进行通信时，虽然会出现报错，但不影响通信。但若是以相同子网，不同源地址或目标地址进行通信时，就将影响通信，对⽹络的实际使⽤造成影响。

为了解决这个问题，修改 as\_switch.py ，在 add\_path 方法中，除了下发原流表外，还会根据源IP和目的IP计算出对应的 /16 子网范围，并添加更高优先级的流表。这一操作可以提高匹配效率，同时可能有助于处理一些特殊的网络需求或优化网络性能。计算 /16 子网范围的方法通过 get\_subnet\_16 方法实现的，该方法将IP地址的前两个字节保留，后两个字节设置为0，并添加 /16 的子网掩码。且无论 gateways 的情况如何，都统一使用子网范围作为源IP和目的IP的参数传入。这样做使得代码逻辑更加一致和清晰，便于理解和维护。

可以看出实验结果如下所示，黑洞问题将不再出现。





# 源代码

## Veriflow.cpp修改部分

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

{

if (graph == NULL) {

return true;

}

if (currentLocation.compare("") == 0) {

return true;

}

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

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

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

updateFaults(packetClass, fp);

if (!isPacketClassInFaults(packetClass)) {

faults.push\_back(packetClass);

}

return false;

}

visited.insert(currentLocation);

if (graph->links.find(currentLocation) == graph->links.end()) {

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

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

updateFaults(packetClass, fp);

if (!isPacketClassInFaults(packetClass)) {

faults.push\_back(packetClass);

}

return false;

}

if (graph->links[currentLocation].empty()) {

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

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

updateFaults(packetClass, fp);

if (!isPacketClassInFaults(packetClass)) {

faults.push\_back(packetClass);

}

return false;

}

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

const list<ForwardingLink>& linkList = graph->links[currentLocation];

list<ForwardingLink>::const\_iterator itr = linkList.begin();

if (lastHop.compare("NULL") == 0 || itr->rule.in\_port == 0) {

// do nothing

}

else {

while (itr != linkList.end()) {

string connected\_hop = network.getNextHopIpAddress(currentLocation, itr->rule.in\_port);

if (connected\_hop.compare(lastHop) == 0) break;

itr++;

}

}

if (itr == linkList.end()) {

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

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

updateFaults(packetClass, fp);

if (!isPacketClassInFaults(packetClass)) {

faults.push\_back(packetClass);

}

return false;

}

if (itr->isGateway == true) {

fprintf(fp, "\n[VeriFlow::traverseForwardingGraph] Packet class reached destination at node %s.\n", currentLocation.c\_str());

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

updateFaults(packetClass, fp);

return true;

}

else {

return this->traverseForwardingGraph(packetClass, graph, itr->rule.nextHop, currentLocation, visited, fp);

}

}

void VeriFlow::updateFaults(const EquivalenceClass& packetClass, FILE\* fp) {

//std::cout << "--- Starting updateFaults ---" << std::endl;

//std::cout << "Before update, faults size: " << faults.size() << std::endl;

int index = 0;

for (auto it = faults.begin(); it != faults.end(); ) {

//std::cout << "\nProcessing fault at index " << index << ":" << std::endl;

//std::cout << "Fault before update: ";

//it->printInfo("");

EquivalenceClass intersection = it->intersection(packetClass);

if (!intersection.isEmpty()) {

//std::cout << "Intersection found:" << std::endl;

//intersection.printInfo("Intersection");

EquivalenceClass diff = it->difference(intersection);

//std::cout << "Difference result:" << std::endl;

//diff.printInfo("Difference");

\*it = diff;

//std::cout << "Fault after update: ";

//it->printInfo("");

if (it->isEmpty()) {

//std::cout << "Fault is empty, removing it from faults list." << std::endl;

it = faults.erase(it);

} else {

//std::cout << "Fault is not empty, keeping it in faults list." << std::endl;

++it;

}

} else {

//std::cout << "No intersection, keeping the fault in faults list." << std::endl;

++it;

}

++index;

}

//std::cout << "After update, faults size: " << faults.size() << std::endl;

//std::cout << "--- Ending updateFaults ---" << std::endl;

}

bool VeriFlow::isPacketClassInFaults(const EquivalenceClass& packetClass)

{

for (const auto& fault : faults) {

if (fault.equals(packetClass)) {

return true;

}

}

return false;

}

## EquivalenceClass.cpp修改部分

EquivalenceClass EquivalenceClass::intersection(const EquivalenceClass& other) const {

uint64\_t newLowerBound[ALL\_FIELD\_INDEX\_END\_MARKER];

uint64\_t newUpperBound[ALL\_FIELD\_INDEX\_END\_MARKER];

bool allEmpty = true;

for (int i = 0; i < ALL\_FIELD\_INDEX\_END\_MARKER; i++) {

newLowerBound[i] = std::max(this->lowerBound[i], other.lowerBound[i]);

newUpperBound[i] = std::min(this->upperBound[i], other.upperBound[i]);

//if (newLowerBound[i] > newUpperBound[i]) {

//uint64\_t temp = newLowerBound[i];

//newLowerBound[i] = newUpperBound[i];

//newUpperBound[i] = temp;

//}

if (newLowerBound[i] == 0 && newUpperBound[i] == 0) {

continue;

} else {

allEmpty = false;

}

}

if (allEmpty) {

return EquivalenceClass();

}

return EquivalenceClass(newLowerBound, newUpperBound);

}

EquivalenceClass EquivalenceClass::difference(const EquivalenceClass& other) const {

std::vector<EquivalenceClass> resultParts;

const EquivalenceClass& intersectionResult = other;

for (int i = 0; i < ALL\_FIELD\_INDEX\_END\_MARKER; i++) {

if (intersectionResult.lowerBound[i] <= intersectionResult.upperBound[i]) {

if (this->lowerBound[i] < intersectionResult.lowerBound[i]) {

EquivalenceClass part = \*this;

part.upperBound[i] = intersectionResult.lowerBound[i] - 1;

resultParts.push\_back(part);

}

if (this->upperBound[i] > intersectionResult.upperBound[i]) {

EquivalenceClass part = \*this;

part.lowerBound[i] = intersectionResult.upperBound[i] + 1;

resultParts.push\_back(part);

}

}

}

if (!resultParts.empty()) {

for (int field = 0; field < ALL\_FIELD\_INDEX\_END\_MARKER; field++) {

std::vector<EquivalenceClass> newResultParts;

while (!resultParts.empty()) {

EquivalenceClass current = resultParts.back();

resultParts.pop\_back();

bool merged = false;

for (auto& part : newResultParts) {

if (current.lowerBound[field] <= part.upperBound[field] &&

current.upperBound[field] >= part.lowerBound[field]) {

part.lowerBound[field] = std::min(current.lowerBound[field], part.lowerBound[field]);

part.upperBound[field] = std::max(current.upperBound[field], part.upperBound[field]);

merged = true;

break;

}

}

if (!merged) {

newResultParts.push\_back(current);

}

}

resultParts = newResultParts;

}

EquivalenceClass mergedResult = resultParts[0];

for (size\_t j = 1; j < resultParts.size(); ++j) {

for (int i = 0; i < ALL\_FIELD\_INDEX\_END\_MARKER; i++) {

mergedResult.lowerBound[i] = std::min(mergedResult.lowerBound[i], resultParts[j].lowerBound[i]);

mergedResult.upperBound[i] = std::max(mergedResult.upperBound[i], resultParts[j].upperBound[i]);

}

}

for (int i = 0; i < ALL\_FIELD\_INDEX\_END\_MARKER; i++) {

if (mergedResult.lowerBound[i] > mergedResult.upperBound[i]) {

uint64\_t temp = mergedResult.lowerBound[i];

mergedResult.lowerBound[i] = mergedResult.upperBound[i];

mergedResult.upperBound[i] = temp;

}

}

return mergedResult;

} else {

return EquivalenceClass();

}

}

bool EquivalenceClass::isEmpty() const {

for (int i = 0; i < ALL\_FIELD\_INDEX\_END\_MARKER; ++i) {

if (this->lowerBound[i] != 0 || this->upperBound[i] != 0) {

return false;

}

## as\_switch.py

from ryu.base import app\_manager

from ryu.base.app\_manager import lookup\_service\_brick

from ryu.controller import ofp\_event

from ryu.controller.handler import MAIN\_DISPATCHER, CONFIG\_DISPATCHER

from ryu.controller.handler import set\_ev\_cls

from ryu.ofproto import ofproto\_v1\_0

from ryu.lib.packet import packet

from ryu.lib.packet import ethernet, arp, ipv4

import utils.flowmod

import utils.ipv4

from utils.flowmod import send\_flow\_mod

from network\_awareness import NetworkAwareness

import json

import os

import math

class RoutingSwitch(app\_manager.RyuApp):

OFP\_VERSIONS = [ofproto\_v1\_0.OFP\_VERSION]

\_CONTEXTS = {

'network\_awareness': NetworkAwareness

}

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(RoutingSwitch, self).\_\_init\_\_(\*args, \*\*kwargs)

self.network\_awareness = kwargs['network\_awareness']

self.dpid\_mac\_port = {}

with open(os.environ["CONFIG"], "r") as f:

self.routing\_cfg = json.load(f)

@set\_ev\_cls(ofp\_event.EventOFPPacketIn, MAIN\_DISPATCHER)

def packet\_in\_handler(self, ev):

msg = ev.msg

dp = msg.datapath

ofp = dp.ofproto

parser = dp.ofproto\_parser

dpid = dp.id

in\_port = msg.in\_port

pkt = packet.Packet(msg.data)

eth\_pkt = pkt.get\_protocol(ethernet.ethernet)

arp\_pkt = pkt.get\_protocol(arp.arp)

ipv4\_pkt = pkt.get\_protocol(ipv4.ipv4)

pkt\_type = eth\_pkt.ethertype

# layer 2 self-learning

dst\_mac = eth\_pkt.dst

src\_mac = eth\_pkt.src

self.dpid\_mac\_port.setdefault(dpid, {})

self.dpid\_mac\_port[dpid][src\_mac] = in\_port

if isinstance(arp\_pkt, arp.arp):

self.handle\_arp(msg, in\_port, dst\_mac, pkt\_type)

if isinstance(ipv4\_pkt, ipv4.ipv4):

self.handle\_ipv4(msg, dpid, in\_port, src\_mac, dst\_mac, ipv4\_pkt.src, ipv4\_pkt.dst, pkt\_type)

def handle\_arp(self, msg, in\_port, dst\_mac, pkt\_type):

dp = msg.datapath

ofp = dp.ofproto

parser = dp.ofproto\_parser

dpid = dp.id

if dst\_mac in self.dpid\_mac\_port[dpid]:

out\_port = self.dpid\_mac\_port[dpid][dst\_mac]

actions = [parser.OFPActionOutput(out\_port)]

out = parser.OFPPacketOut(

datapath=dp, buffer\_id=msg.buffer\_id, in\_port=in\_port, actions=actions, data=msg.data)

dp.send\_msg(out)

else:

# send to the switch port which linked hosts

for d, ports in self.network\_awareness.port\_info.items():

for p in ports:

# except the source host

if d == dpid and p == in\_port:

continue

dp = self.network\_awareness.switch\_info[d]

actions = [parser.OFPActionOutput(p)]

out = parser.OFPPacketOut(

datapath=dp, buffer\_id=msg.buffer\_id, in\_port=ofp.OFPP\_CONTROLLER, actions=actions, data=msg.data)

dp.send\_msg(out)

def handle\_ipv4(self, msg, dpid, in\_port, src\_mac, dst\_mac, src\_ip, dst\_ip, pkt\_type):

print(f"Packet to {dpid}")

parser = msg.datapath.ofproto\_parser

switch\_net = self.routing\_cfg["switch\_nets"][dpid]

srcnet = None

dstnet = None

for net in self.routing\_cfg["gateways"]:

if utils.ipv4.in\_net(net, src\_ip):

srcnet = net

break

if utils.ipv4.in\_net(switch\_net, dst\_ip):

# Local switching. Install shortest paths.

dstnet = switch\_net

gateways = None

else:

# Routing. Go to the closest gateway

for dst\_candidate in self.routing\_cfg["gateways"][switch\_net]:

if utils.ipv4.in\_net(dst\_candidate, dst\_ip):

dstnet = dst\_candidate

gateways = self.routing\_cfg["gateways"][switch\_net][dstnet]

if not srcnet or not dstnet:

print("src / dst not recognized, unable to forward.")

return

# add\_path should be called only from switch to switch / host.

def add\_path(route, dl\_src, dl\_dst, nw\_src, nw\_dst, priority=5):

port\_path = []

for i in range(len(route) - 1):

out\_port = self.network\_awareness.link\_info[(route[i], route[i + 1])]

port\_path.append((route[i], out\_port))

self.show\_path(route[0], route[-1], port\_path)

for node in port\_path:

waypoint\_dpid, out\_port = node

send\_flow\_mod(waypoint\_dpid, dl\_src, dl\_dst, nw\_src, nw\_dst, None, out\_port, priority)

src\_net\_16 = self.get\_subnet\_16(nw\_src)

dst\_net\_16 = self.get\_subnet\_16(nw\_dst)

high\_priority = 10

send\_flow\_mod(waypoint\_dpid, dl\_src, dl\_dst, src\_net\_16, dst\_net\_16, None, out\_port, high\_priority)

# return the output port of current switch.

return port\_path[0][1]

if gateways is None:

# Handle packet switching within AS.

dpid\_path = self.network\_awareness.shortest\_path(dpid, dst\_ip)

if not dpid\_path:

return

# get port path: h1 -> in\_port, s1, out\_port -> h2

out\_port = add\_path(dpid\_path, None, None, srcnet, dstnet)

else:

# Handle inter-AS packet switching.

if dpid in gateways:

# first, if self is a gateway, send to the peer

peer = self.routing\_cfg["peers"][str(dpid)][dstnet]

route = [dpid, peer]

out\_port = add\_path(route, None, None, srcnet, dstnet)

else:

# otherwise, send to the closest gateway

min\_delay = math.inf

min\_gw = None

for gw in gateways:

delay = self.network\_awareness.shortest\_path\_length(dpid, gw)

if delay < 0:

continue

if delay < min\_delay:

min\_delay = delay

min\_gw = gw

if min\_gw is None:

# no way to gateway

return

dpid\_path = self.network\_awareness.shortest\_path(dpid, min\_gw)

if not dpid\_path:

return

out\_port = add\_path(dpid\_path, None, None, srcnet, dstnet)

# after installing the path, send the packet

dp = self.network\_awareness.switch\_info[dpid]

actions = [parser.OFPActionOutput(out\_port)]

out = parser.OFPPacketOut(

datapath=dp, buffer\_id=msg.buffer\_id, in\_port=in\_port, actions=actions, data=msg.data)

dp.send\_msg(out)

def get\_subnet\_16(self, ip):

parts = ip.split('.')

return f"{parts[0]}.{parts[1]}.0.0/16"

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

self.logger.info('path: {} -> {}'.format(src, dst))

path = str(src) + ' -> '

for node in port\_path:

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

path += str(dst)

self.logger.info(path)

self.logger.info('\n')