**LAB4-VeriFlow**

**（一）热身：转发环路**

**观察转发环路**

1. 启动拓扑

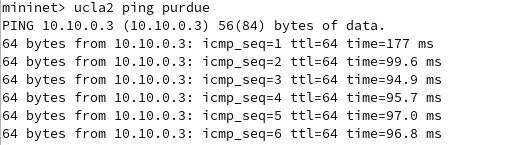
sudo ./simple.py

2. 启动控制程序

TOPO=simple.txt CONFIG=simple.config.json uv run ryu-manager ryu.app.ofctl\_rest as\_switch.py

3.在拓扑中ucla2 ping purdue建⽴连接

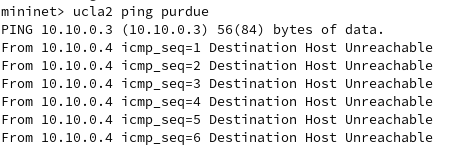
mininet> ucla2 ping purdue



控制器计算最短路径并下发流表项，数据包途经s6(usla2) -> s3(illinois) -> s5(purdue)

4. 下发从illinois途经wisconsin到达ucla2的路径之后，ucla2 ping purdue失败

uv run ./gen\_loop.py

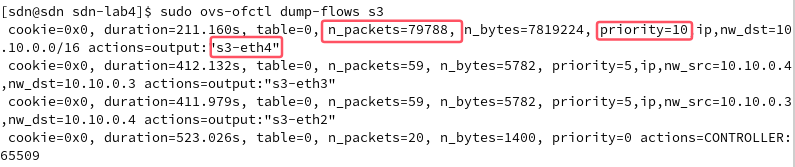


拓扑中产生了路由环路，无法ping通

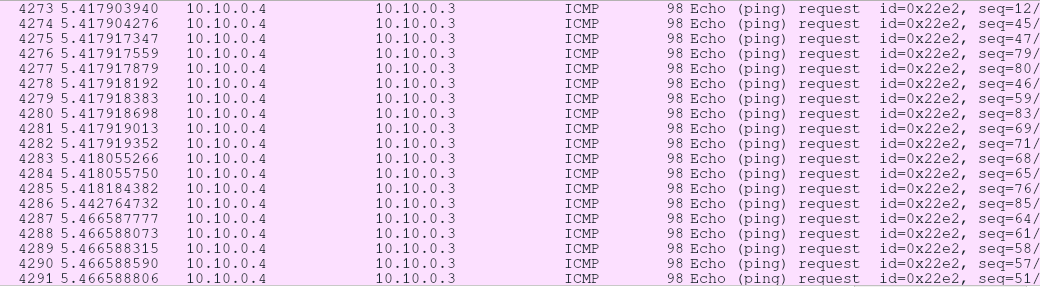
5. 查看路径上某⼀交换机，如illinois的流表，发现匹配某⼀条流表的数据包数⽬异常增加

也可打开wireshark观察该端⼝，发现不断增加的ICMP Request报⽂

sudo ovs-ofctl dump-flows s3



交换机s3中的新流表项优先级大于先前最短路径的优先级，从10.10.0.0收到的数据包将从 4端口转发出



s3交换机捕获到了大量ICMP报文

**使⽤VeriFlow验证环路问题**

**实验内容**

1. 输出每次影响EC的数量。

在VeriFlow.cpp的verifyRule函数中，把打印ecCount的代码打印到日志文件即可

|  |
| --- |
| else{  // fprintf(stdout, "\n");  // fprintf(stdout, "[VeriFlow::verifyRule] ecCount: %lu\n", ecCount);  fprintf(fp, "[VeriFlow::verifyRule] ecCount: %lu\n", ecCount);  } |

2. 打印出环路路径的信息。

VeriFlow::traverseForwardingGraph()函数遍历某个特定EC的转发图，验证是否存在环路或⿊洞，意味着该函数可以保存环路路径信息，可以加上一个参数vector<string>& path

在VeriFlow.h中修改traverseForwardingGraph函数的定义

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

在VeriFlow.cpp中修改traverseForwardingGraph函数的定义，并且所有该函数的调用需要添加一个参数path

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

在VeriFlow.cpp的traverseForwadingGraph函数中加入记录路径和输出环路路径的代码

|  |
| --- |
| if(visited.find(currentLocation) != visited.end()) //当前地址已经被遍历过，说明产生环路  {  // 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());  for (int i=0;i<path.size();++i){ //打印环路路径  fprintf(fp,"%s -> ",path[i].c\_str());  }  fprintf(fp,"%s\n",currentLocation.c\_str());  ...  }  path.push\_back(currentLocation); //当前地址没有被遍历过，记录路径  visited.insert(currentLocation); //加入被遍历过的数组 |

3. 进⼀步打印出环路对应的EC的相关信息。仅展示源MAC、⽬的MAC和TCP/IP五元组（即源IP、⽬的IP、协议、源端⼝、⽬的端⼝）。

在EquivalenceClass.cpp中修改to\_String函数，添加需要打印的信息

|  |
| --- |
| string EquivalenceClass::toString() const  {  char buffer[1024];  sprintf(buffer, "[EquivalenceClass] \ndl\_src (%s, %s) \ndl\_dst (%s, %s)\n",  ::getMacValueAsString(this->lowerBound[DL\_SRC]).c\_str(),  ::getMacValueAsString(this->upperBound[DL\_SRC]).c\_str(),  ::getMacValueAsString(this->lowerBound[DL\_DST]).c\_str(),  ::getMacValueAsString(this->upperBound[DL\_DST]).c\_str());  string retVal = buffer;  sprintf(buffer, "nw\_src (%s, %s) \nnw\_dst (%s, %s)\n",  ::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());  retVal += buffer;  sprintf(buffer, "nw\_proto (%lu, %lu) \ntp\_src (%lu, %lu) \ntp\_dst (%lu, %lu)",  this->lowerBound[NW\_PROTO],  this->upperBound[NW\_PROTO],  this->lowerBound[TP\_SRC],  this->upperBound[TP\_SRC],  this->lowerBound[TP\_DST],  this->upperBound[TP\_DST]);  retVal += buffer;  return retVal;  } |

1.编译VeriFlow.

其中， -j$(nproc)为指定编译线程数，可以不加

make -C veriflow -j$(nproc)

2. 在⾃定义端⼝开启远程控制器，运⾏AS控制器程序

TOPO=simple.txt CONFIG=simple.config.json uv run ryu-manager ryu.app.ofctl\_rest as\_switch.py --ofp-tcp-listen-port 1024

3. 运⾏VeriFlow的proxy模式

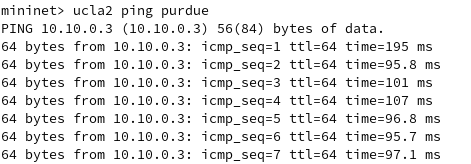
veriflow/VeriFlow 6633 127.0.0.1 1024 simple.txt veriflow.log

4. 启动拓扑

sudo ./simple.py

5. 在拓扑中ucla2 ping purdue建⽴连接

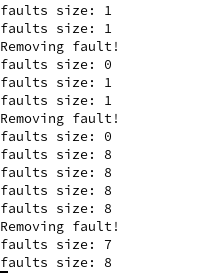
mininet> ucla2 ping purdue



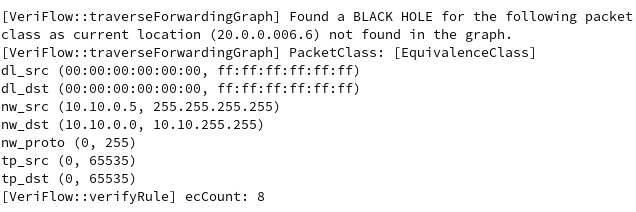
6.下发从illinois途经wisconsin到达ucla2的路径，在log⽂件中观察VeriFlow检测到的环路信息

uv run ./gen\_loop.py



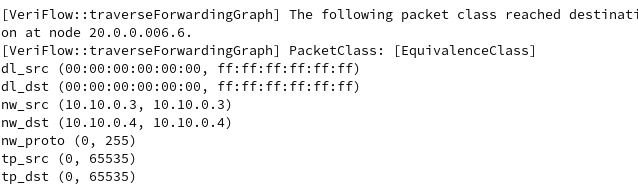


log文件中VeriFlow记录的信息如下：

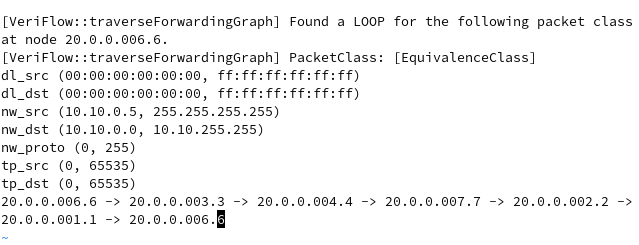


影响EC的数量为8

由于控制器无法同时下发多个流表项，VeriFlow在对某条流表项进行检查时发现数据包匹配该流表项后转发到下一个交换机，但此时下一个交换机中还没有处理该数据包的流表项，因此 VeriFlow记录黑洞信息



当某条流表项能够使得数据包被转发到主机时，VeriFlow记录目的主机可达信息



VeriFlow检测到环路，s6 -> s3 -> s4 -> s7 -> s2 -> s1-> s6， 即ucla2 -> illinois -> wisconsin -> usc2 -> usc1 -> ucla1 -> ucla2

**实验：⿊洞和VeriFlow的局限**

**建⽴转发⿊洞**

1. 在⾃定义端⼝开启远程控制器，运⾏AS控制器程序

TOPO=simple.txt CONFIG=simple.config.json uv run ryu-manager ryu.app.ofctl\_rest as\_switch.py --ofp-tcp-listen-port 1024

2. 运⾏VeriFlow的proxy模式：

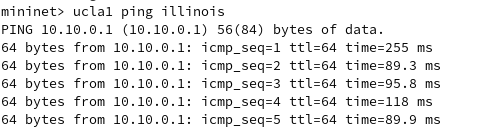
veriflow/VeriFlow 6633 127.0.0.1 1024 simple.txt veriflow.log

3. 启动拓扑

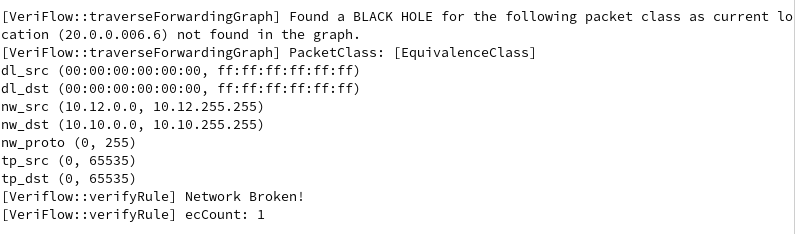
sudo ./simple.py

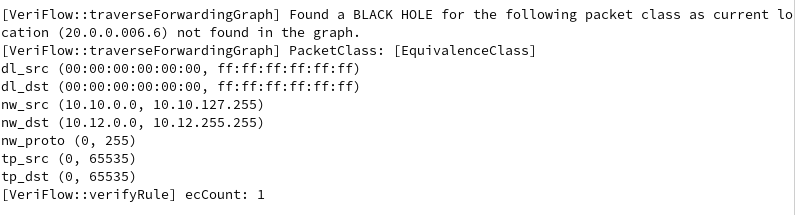
4. 建⽴转发路径

mininet> ucla1 ping illinois



此时，两者可以正常ping通，但是VeriFlow提示出现了⿊洞。





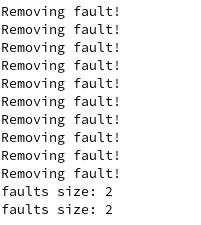
由于控制器无法同时下发多个流表项，VeriFlow在对某条流表项进行检查时发现数据包匹配该流表项后转发到下一个交换机，但此时下一个交换机中还没有处理该数据包的流表项，VeriFlow检测到某个等价类的数据包无法到达目的地时，会记录一个黑洞错误信息。

可以手动添加流表项，在网络中建立从源节点到目的节点的完整转发路径，从而消除由于路径不完整而导致的数据包黑洞。

你很快找出了⿊洞的原因，并⼿动添加了⼏条流表项来解决这个问题。

uv run ./fix\_path.py

然⽽，你发现即使这个问题理论上解决了，但是，VeriFlow的发现的⽹络错误没有消失！



这是因为，在VeriFlow代码中，对新规则影响错误数量的计算有⼀些错误：

代码仅仅考虑了新规则的等价类完全包含错误的等价类的情况；此时，如果⼀个新规则修复了部分路径的错误，但其等价类只覆盖了部分之前的错误，那么旧的错误会依然存在；导致错误数量的计算可能有误。

解决方法：删除packetClass影响的已有fault时，如果一个fault和packetClass有交集，那么为保持一致性，该fault要将和这个packetClass交集的部分删除；若删除交集后该fault是空的（即没有覆盖到任何一个域），则将这个fault去除。

代码修改：

在EquivalenceClass类中添加intersects方法，判断两个等价类是否相交

1. 在EquivalenceClass.h中添加方法声明：

bool intersects(const EquivalenceClass& other) const;

1. 在EquivalenceClass.cpp中实现intersects方法：

原本subsumes函数只考虑了全包含的情况

|  |
| --- |
| bool EquivalenceClass::subsumes(const EquivalenceClass &other) const  {  for(int i = 0; i < ALL\_FIELD\_INDEX\_END\_MARKER; i++)  {  if((this->lowerBound[i] <= other.lowerBound[i] && this->upperBound[i] >= other.upperBound[i])) {} //当前等价类地址下界<=另一等价类下界，同时当前等价类地址上界>=另一等价类上界，即为当前等价类覆盖了另一等价类  else {  return false;  }  }  return true;  } |

添加intersects方法，计算两等价类交集

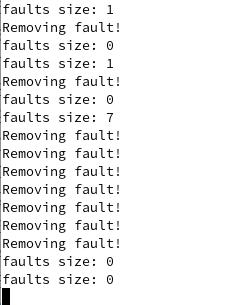
|  |
| --- |
| bool EquivalenceClass::intersects(const EquivalenceClass& other) const {  for (int i = 0; i < ALL\_FIELD\_INDEX\_END\_MARKER; i++) {  if (this->upperBound[i] < other.lowerBound[i] || this->lowerBound[i] > other.upperBound[i]) { //当前等价类地址的上界<另一等价类地址的下界，或者当前等价类地址的下界>另一等价类的上界，即两等价类没有交集  return false;  }  }  return true; //若有交集，返回true  } |

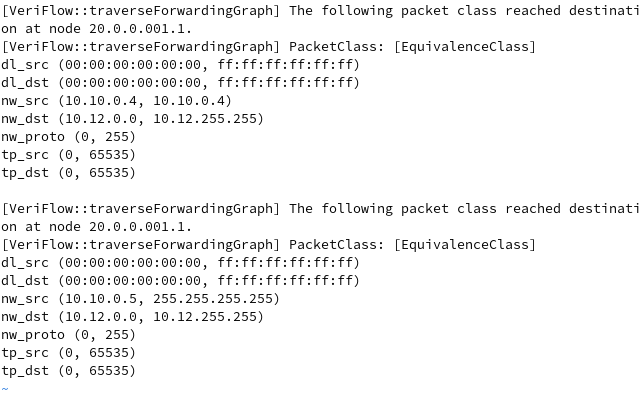
3. 修改VeriFlow.cpp中的错误处理逻辑：用intersects方法替换subsumes

|  |
| --- |
| for (int i = 0; i < faults.size();) {  if (packetClass.intersects(faults[i])) { //如果先前等价类中某个地址与新等价类有交集  faults.erase(faults.begin() + i); //把该地址从先前等价类faults中删除  } else {  i++; //没有交集则跳过  }  }  faults.push\_back(packetClass); //如果有新错误，加入错误列表 |

|  |
| --- |
| for (int i = 0; i < faults.size();) {  if (packetClass.intersects(faults[i])) {  fprintf(stderr, "Removing fault!\n");  faults.erase(faults.begin() + i);  } else {  i++;  }  } |

修改后





VeriFlow中不再检测到黑洞

**选做题：⽹络验证和分析**

1. 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。

由于控制器无法同时下发多个流表项，VeriFlow在对某条流表项进行检查时发现数据包匹配该流表项后转发到下一个交换机，但此时下一个交换机中还没有处理该数据包的流表项，VeriFlow检测到某个等价类的数据包无法到达目的地时，会记录一个黑洞错误信息。

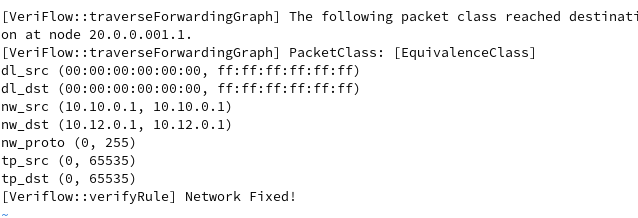
不会影响实际网络使用，因为误报源于检测机制，但控制器能正确安装流表项，实际转发正常。

解决方案：

集中收集流表项并批量下发

收集路径上所有需要下发的流表项到flow\_mods列表中，再统一下发

|  |
| --- |
| def add\_path(route, dl\_src, dl\_dst, nw\_src, nw\_dst, dl\_type, priority=5):  port\_path = []  flow\_mods = []  for i in range(len(route) - 1):  current\_dpid = route[i]  next\_dpid = route[i + 1]  out\_port = self.network\_awareness.link\_info[(current\_dpid, next\_dpid)]  port\_path.append((current\_dpid, out\_port))  flow\_mods.append((current\_dpid, dl\_src, dl\_dst, nw\_src, nw\_dst, dl\_type, out\_port, priority)) //收集所有流表项到列表  self.show\_path(route[0], route[-1], port\_path)    for fm in flow\_mods:  send\_flow\_mod(\*fm) //发送所有流表项    return port\_path[0][1] if port\_path else None |



VeriFlow不再检测到黑洞