Lab4-VeriFlow

（一）实验前准备

（1）观察转发环路

①启动拓扑：

sudo python Arpanet19723.py

②启动最短路径的控制程序：

ryu-manager ofctl_rest.py shortest_path.py --observe-links

③在拓扑中 SDC ping MIT 建立连接：

mininet> SDC ping MIT

```
mininet> SDC ping MIT
PING 10.0.0.12 (10.0.0.12) 56(84) bytes of data.
64 bytes from 10.0.0.12: icmp_seq=3 ttl=64 time=211 ms
64 bytes from 10.0.0.12: icmp_seq=4 ttl=64 time=241 ms
64 bytes from 10.0.0.12: icmp_seq=5 ttl=64 time=260 ms
64 bytes from 10.0.0.12: icmp_seq=6 ttl=64 time=318 ms
64 bytes from 10.0.0.12: icmp_seq=7 ttl=64 time=237 ms
^C
--- 10.0.0.12 ping statistics ---
7 packets transmitted, 5 received, 28% packet loss, time 6089ms
rtt min/avg/max/mdev = 211.613/254.006/318.450/35.813 ms
```

```
test@sdnexp:~/Desktop/sdn_exp/sdn_exp_4$ ryu-manager ofctl_rest.py shortest_path.py
 --observe-links
loading app ofctl_rest.py
loading app shortest_path.py
loading app ryu.controller.ofp_handler
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
creating context wsgi
instantiating app None of NetworkAwareness
creating context network_awareness
instantiating app shortest_path.py of ShortestPath
instantiating app ryu.topology.switches of Switches
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ofctl_rest.py of RestStatsApi
(27676) wsgi starting up on http://0.0.0.0:8080
path: 10.0.0.18 -> 10.0.0.12
10.0.0.18 -> 1:s15:3 -> 3:s22:4 -> 4:s23:2 -> 3:s1:2 -> 2:s25:1 -> 10.0.0.12
```

此时，SDC 与 MIT 之间可以 ping 通，且选择了跳数最少的路由。

④下发从 UTAH 途经 TINKER 到达 ILLINOIS 的路径：

sudo python waypoint_path.py

```
test@sdnexp:~/Desktop/sdn_exp/sdn_exp_4$ sudo python waypoint_path.py
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
install waypoint path: 23 -> 1
23 -> 4:s22:2 -> 2:s9:3 -> 3:s16:2 -> 3:s7:2 -> 3:s25:2 -> 1
```

下发新的转发路径后，拓扑中产生了路由环路。交换机 s22 中的新流表项优先级大于先前最短路径的优先级，于是从 port4 收到的数据包将从 port2 转发出去而不是先前的 port3；同理，在交换机 s25 中，从 port2 收到的数据包将从 port3 转发出去而不是先前的 port1。由此产生的路由环路为：

s22 -> s23 -> s1 -> s25 -> s7 -> s16 -> s9 -> s22

⑤在拓扑中 SDC ping MIT 建立连接：

mininet> SDC ping MIT

```
mininet> SDC ping MIT
PING 10.0.0.12 (10.0.0.12) 56(84) bytes of data.
^C
--- 10.0.0.12 ping statistics ---
8 packets transmitted, 0 received, 100% packet loss, time 7182ms
```

此时，由于路由环路的存在，SDC 无法 ping 通 MIT。

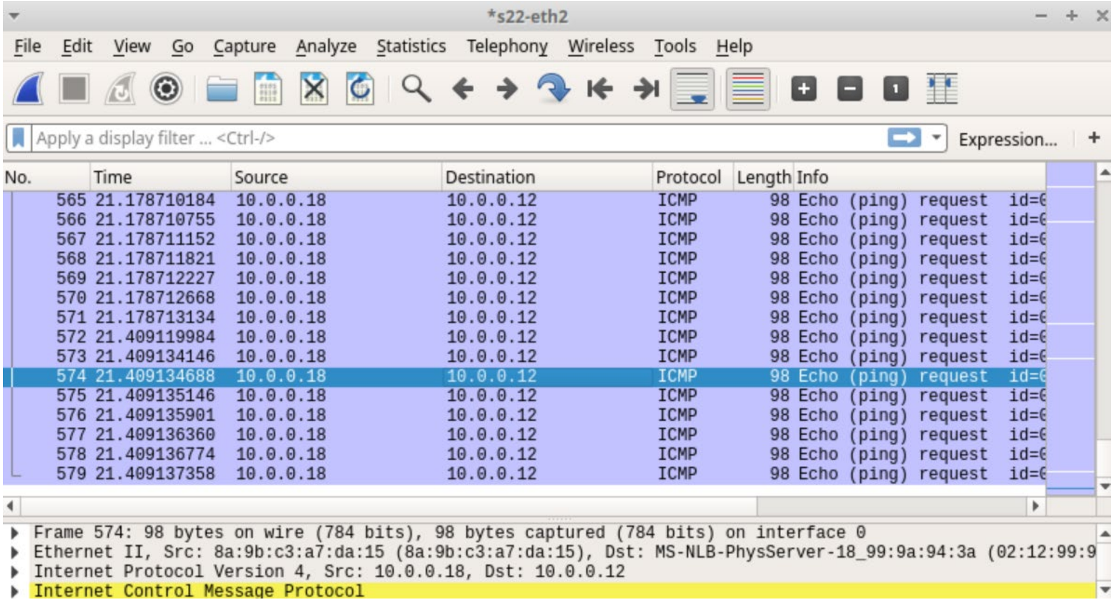
⑥查看交换机 s22 的流表：

sudo ovs-ofctl dump-flows s22

```
test@sdnexp:~/Desktop/sdn_exp/sdn_exp_4$ sudo ovs-ofctl dump-flows s22
 cookie=0x0, duration=797.675s, table=0, n_packets=589, n_bytes=35340, priority=65535,d
l_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:60
 cookie=0x0, duration=685.798s, table=0, n_packets=12, n_bytes=1176, priority=1,ip,in_p
ort="s22-eth3",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s22-eth4"
 cookie=0x0, duration=685.793s, table=0, n_packets=5, n_bytes=490, priority=1,ip,in_por
t="s22-eth4",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s22-eth3"
 cookie=0x0, duration=338.570s, table=0, n_packets=0, n_bytes=0, priority=10,ip,in_port
="s22-eth4",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s22-eth2"
 cookie=0x0, duration=338.566s, table=0, n_packets=8127, n_bytes=796446, priority=10,ip
,in_port="s22-eth2",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s22-eth4"
 cookie=0x0, duration=797.675s, table=0, n_packets=58, n_bytes=5728, priority=0 actions
=CONTROLLER:65509
```

观察发现，交换机 s22 中从 port2 进入的数据包的流表项匹配次数非常多，说明了路由环路的存在。

⑦使用 Wireshark 观察 s22-eth2 端口：

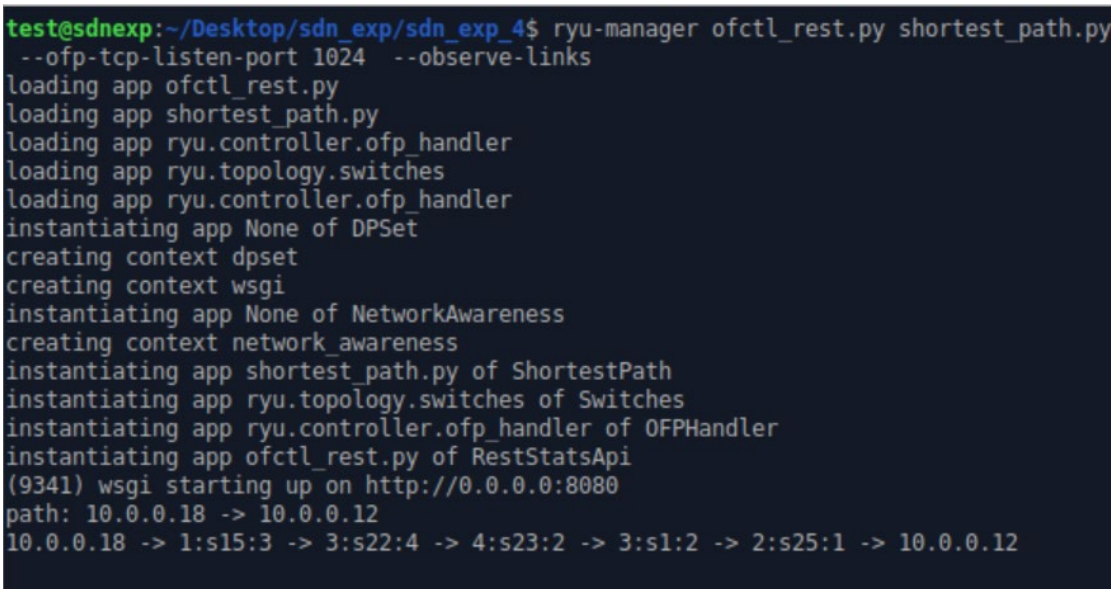


使用Wireshark观察交换机s22的port2，发现了大量源IP地址为10.0.0.18（SDC）、目的IP地址为10.0.0.12（MIT）的ICMP请求报文，这也从另一方面说明了路由环路的存在。

（2）使用VeriFlow

①在自定义端口开启远程控制器，运行最短路程序：

ryu-manager ofctl_rest.py shortest_path.py --ofp-tcp-listen-port 1024 --observe-links



②运行VeriFlow的proxy模式：

./VeriFlow 6633 127.0.0.1 1024 Arpanet19723.txt log_file.txt

```
test@sdnexp:~/Desktop/sdn_exp/sdn_exp_4/BEADS/veriflow/VeriFlow$ ./VeriFlow 6633 127.0.0.1 1024 ../../../Arpanet19723.txt ../../../log_file.txt
id 125 ipAddress 10.0.0.25 endDevice 1 port 0 nextHopIpAddress 20.0.0.23

id 122 ipAddress 10.0.0.22 endDevice 1 port 0 nextHopIpAddress 20.0.0.20

id 120 ipAddress 10.0.0.20 endDevice 1 port 0 nextHopIpAddress 20.0.0.21

id 117 ipAddress 10.0.0.17 endDevice 1 port 0 nextHopIpAddress 20.0.0.13

id 116 ipAddress 10.0.0.16 endDevice 1 port 0 nextHopIpAddress 20.0.0.11

id 115 ipAddress 10.0.0.15 endDevice 1 port 0 nextHopIpAddress 20.0.0.8

id 118 ipAddress 10.0.0.18 endDevice 1 port 0 nextHopIpAddress 20.0.0.15

id 114 ipAddress 10.0.0.14 endDevice 1 port 0 nextHopIpAddress 20.0.0.12

id 113 ipAddress 10.0.0.13 endDevice 1 port 0 nextHopIpAddress 20.0.0.2
```

后续生成转发环路的过程与（1）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：

```
[VeriFlow::traverseForwardingGraph] Found a BLACK HOLE for the following packet class as there is no outgoing link at current location (20.0.0.25).
[VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src (0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), dl_dst (1652522221583-01:80:c2:00:00:0f, 281474976710655-ff:ff:ff:ff:ff:ff), nw_src (167772160-10.0.0.0, 167772415-10.0.0.255), nw_dst (167772160-10.0.0.0, 167772415-10.0.0.255), Field 0 (0, 65535), Field 1 (0, 281474976710655), Field 2 (1652522221583, 281474976710655), Field 3 (2048, 2048), Field 4 (0, 4095), Field 5 (0, 7), Field 6 (0, 1048575), Field 7 (0, 7), Field 8 (167772160, 167772415), Field 9 (167772160, 167772415), Field 10 (0, 255), Field 11 (0, 63), Field 12 (0, 65535), Field 13 (0, 65535)
```

由于控制器无法同时下发多个流表项，因此 VeriFlow 在对某条流表项进行检查时可能会发现：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。

```
[VeriFlow::traverseForwardingGraph] The following packet class reached destination at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src (0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), dl_dst (0-00:00:00:00:00:00, 1652522221581-01:80:c2:00:00:0d), nw_src (167772160-10.0.0.0, 167772415-10.0.0.255), nw_dst (167772160-10.0.0.0, 167772415-10.0.0.255), Field 0 (0, 65535), Field 1 (0, 281474976710655), Field 2 (0, 1652522221581), Field 3 (2048, 2048), Field 4 (0, 4095), Field 5 (0, 7), Field 6 (0, 1048575), Field 7 (0, 7), Field 8 (167772160, 167772415), Field 9 (167772160, 167772415), Field 10 (0, 255), Field 11 (0, 63), Field 12 (0, 65535), Field 13 (0, 65535)
```

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

```
[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src (0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), dl_dst (1652522221582-01:80:c2:00:00:0e, 1652522221582-01:80:c2:00:00:0e), nw_src (167772160-10.0.0.0, 167772415-10.0.0.255), nw_dst (167772160-10.0.0.0, 167772415-10.0.0.255), Field 0 (0, 65535), Field 1 (0, 281474976710655), Field 2 (1652522221582, 1652522221582), Field 3 (2048, 2048), Field 4 (0, 4095), Field 5 (0, 7), Field 6 (0, 1048575), Field 7 (0, 7), Field 8 (167772160, 167772415), Field 9 (167772160, 167772415), Field 10 (0, 255), Field 11 (0, 63), Field 12 (0, 65535), Field 13 (0, 65535)
```

在下发新的流表项时，VeriFlow 发现新流表项将会导致路由环路产生，便在日志文件中记录路由环路信息。

（二）基础实验部分

①EC 数目的打印：

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，将其打印到日志文件即可。

修改代码如下：

```
    fprintf(fp, "\n[VeriFlow::verifyRule] verifying this rule: %s\n", 
rule.toString().c_str());
    ……
    ecCount = vFinalPacketClasses.size();
    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);
        fprintf(fp, "[VeriFlow::verifyRule] ecCount: %lu\n", ecCount);
    }
```

日志文件截图如下：

```
[VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dlSrcAddr: 00:00:00:00:00:00,
dlSrcAddrMask: 0:0:0:0:0:0, dlDstAddr: 00:00:00:00:00:00, dlDstAddrMask: 0:0:0:0:0:0,
nwSrcAddr: 10.0.0.0, nwSrcAddrMask: 255.255.255.0, nwDstAddr: 10.0.0.0, nwDstAddrMask:
255.255.255.0, location: 20.0.0.25, nextHop: 20.0.0.7, in_port: 2, priority: 10, wildcards:
3279086, in_port: 0, dl_type: 2048, dl_vlan: 0, dl_vlan_pcp: 0, mpls_label: 0, mpls_tc: 0,
nw_proto: 0, nw_tos: 0, tp_src: 0, tp_dst: 0
[VeriFlow::verifyRule] ecCount: 3
```

②环路路径的打印：

VeriFlow::traverseForwardingGraph()遍历某个特定 EC 的转发图，验证是否存在环路或黑洞。该函数中变量 visited 负责记录遍历过的节点，若当前节点存在于 visited 中，则说明出现了环路。由于 visited 是 unordered_set 类型，使用哈希技术对元素进行无序存储，为了保证搜索的速度，可以增加一个变量 vector<string> loop_path 用于记录环路。

修改代码如下：

```
bool VeriFlow::traverseForwardingGraph(const EquivalenceClass& packetClass,
ForwardingGraph* graph, const string& currentLocation, const string& lastHop,
unordered_set<string> visited, FILE* fp, vector<string> loop_path)
{
    ……
    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());

        fprintf(fp, "[VeriFlow::traverseForwardingGraph] Loop path is:\n");
        for(unsigned int i = 0; i < loop_path.size(); i++)
            fprintf(fp, "%s -> ", loop_path[i].c_str());
        fprintf(fp, "%s\n", currentLocation.c_str());

        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);
    loop_path.push_back(currentLocation);
    ……
        return this->traverseForwardingGraph(packetClass, graph, itr->rule.nextHop,
currentLocation, visited, fp, loop_path);
    }
}
```

此外，还需要修改对应头文件中的函数定义，且调用该函数时应传入参数vector<string> loop_path。

日志文件截图如下：

```
[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src (0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), dl_dst (1652522221583-01:80:c2:00:00:0f, 281474976710655-ff:ff:ff:ff:ff:ff), nw_src (167772160-10.0.0.0, 167772415-10.0.0.255), nw_dst (167772160-10.0.0.0, 167772415-10.0.0.255), Field 0 (0, 65535), Field 1 (0, 281474976710655), Field 2 (1652522221583, 281474976710655), Field 3 (2048, 2048), Field 4 (0, 4095), Field 5 (0, 7), Field 6 (0, 1048575), Field 7 (0, 7), Field 8 (167772160, 167772415), Field 9 (167772160, 167772415), Field 10 (0, 255), Field 11 (0, 63), Field 12 (0, 65535), Field 13 (0, 65535)
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25
```

③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。

EC 的基本信息打印在函数 VeriFlow::traverseForwardingGraph()中通过 packetClass.toString().c_str()实现，现仿照 EquivalenceClass::toString()函数，向 EquivalenceClass 类中添加 TcpIptoString()函数。

修改代码如下：

```
string EquivalenceClass::TcpIpToString() const
{
    char buffer[1024];
    sprintf(buffer, "nw_src(%s-%s), nw_dst(%s-%s)",
        ::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());

    string retVal = buffer;
    retVal += ", ";
    sprintf(buffer, "nw_proto(%lu-%lu)", this->lowerBound[NW_PROTO],
this->upperBound[NW_PROTO]);
    retVal += buffer;
    retVal += ", ";
    sprintf(buffer, "tp_src(%lu-%lu)", this->lowerBound[TP_SRC],
this->upperBound[TP_SRC]);
    retVal += buffer;
    retVal += ", ";
    sprintf(buffer, "tp_dst(%lu-%lu)", this->lowerBound[TP_DST],
this->upperBound[TP_DST]);
    retVal += buffer;
    return retVal;
```

```
}
```

此外，还需要在头文件中向 EquivalenceClass 类添加 TcpIptoString()函数的定义，并修改先前的函数调用。

日志文件截图如下：

```
[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node
20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src(10.0.0.0-10.0.0.255),
nw_dst(10.0.0.0-10.0.0.255), nw_proto(0-255), tp_src(0-65535), tp_dst(0-65535)
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 ->
20.0.0.25
```

④分析原始代码与补丁代码的区别，思考为何需要添加补丁：

```
diff --git a/veriflow/VeriFlow/OpenFlowProtocolMessage.cpp b/veriflow/VeriFlow/OpenFlowProtocolMessage.cpp
index ac5084d..a081d3b 100644
--- a/veriflow/VeriFlow/OpenFlowProtocolMessage.cpp
+++ b/veriflow/VeriFlow/OpenFlowProtocolMessage.cpp
@@ -292,10 +292,8 @@ void OpenFlowProtocolMessage::processFlowRemoved(const char* data, ProxyConnecti
        rule.type = FORWARDING;
        rule.wildcards = ntohl(ofr->match.wildcards);

-       rule.fieldValue[IN_PORT] = "0";//::convertIntToString(ntohs(ofr->match.in_port));
-       rule.fieldMask[IN_PORT] = "0";//((rule.wildcards == OFPFW_ALL) || ((rule.wildcards & OFPFW_IN_PORT) !=
0)) ? "0" : "65535";
-
-       rule.in_port = ntohs(ofr->match.in_port);
+       rule.fieldValue[IN_PORT] = ::convertIntToString(ntohs(ofr->match.in_port));
+       rule.fieldMask[IN_PORT] = ((rule.wildcards == OFPFW_ALL) || ((rule.wildcards & OFPFW_IN_PORT) != 0)) ?
"0" : "65535";

        rule.fieldValue[DL_SRC] = ::getMacValueAsString(ofr->match.dl_src);
        rule.fieldMask[DL_SRC] = ((rule.wildcards == OFPFW_ALL) || ((rule.wildcards & OFPFW_DL_SRC) != 0)) ? "0
:0:0:0:0:0" : "FF:FF:FF:FF:FF:FF";
```

补丁代码中将 fieldMask[IN_PORT]设置成 0，屏蔽了匹配域 IN_PORT，其目的是为了防止错误环路的产生：

对于形如 x:s1:y <-> z:s2:x 的链路，在对匹配域 IN_PORT=x 的规则进行验证时，将在两个交换机之间形成环路。

```
diff --git a/veriflow/VeriFlow/Rule.cpp b/veriflow/VeriFlow/Rule.cpp
index 847d902..a0ec591 100644
--- a/veriflow/VeriFlow/Rule.cpp
+++ b/veriflow/VeriFlow/Rule.cpp
@@ -36,7 +36,6 @@ Rule::Rule()

        this->location = "";
        this->nextHop = "";
-       this->in_port = 65536;
        this->priority = INVALID_PRIORITY;
        // this->outPort = OFPP_NONE;
 }
```

补丁代码向 Rule 中增加了 in_port 属性。

```
@@ -1045,9 +1043,7 @@ bool VeriFlow::verifyRule(const Rule& rule, int command, double& updateTime, dou
        for(unsigned int i = 0; i < vGraph.size(); i++)
        {
                unordered_set< string > visited;
-               string lastHop = network.getNextHopIpAddress(rule.location,rule.in_port);
-               // fprintf(fp, "start traversing at: %s\n", rule.location.c_str());
-               if(!this->traverseForwardingGraph(vFinalPacketClasses[i], vGraph[i], rule.location, lastHop, vi
sited, fp)) {
+               if(!this->traverseForwardingGraph(vFinalPacketClasses[i], vGraph[i], rule.location, visited, fp
)) {
                        ++currentFailures;
                }
        }
```

补丁代码在 VeriFlow::verifyRule 中增加了 lastHop 变量，并修改了 VeriFlow::traverseForwardingGraph()函数定义，新增了参数 lastHop，用于后续对环路与黑洞的检测。

```
@@ -1163,33 +1157,6 @@ bool VeriFlow::traverseForwardingGraph(const EquivalenceClass& packetClass, Forw

        const list< ForwardingLink >& linkList = graph->links[currentLocation];
        list< ForwardingLink >::const_iterator itr = linkList.begin();
-       // input_port as a filter
-       if(lastHop.compare("NULL") == 0 || itr->rule.in_port == 65536){
-               // 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()){
-               // Found a black hole.
-               fprintf(fp, "\n");
-               fprintf(fp, "[VeriFlow::traverseForwardingGraph] Found a BLACK HOLE for the following packet cl
ass as there is no outgoing link at current location (%s).\n", currentLocation.c_str());
-               fprintf(fp, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n", packetClass.toString().c_s
tr());
-               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;
-       }

        if(itr->isGateway == true)
        {
```

补丁代码完善了在转发图中获取下一跳地址的方法。在原始代码中，首先对当前交换机中的规则按照优先级进行排序，然后选择第一条规则以获取下一跳的地址。但这样可能导致下一跳与上一跳相同，从而判断出错误的环路。

修改后，补丁代码首先利用 itr->rule.in_port 查找该端口相连的交换机，若查找结果与上一跳相同，说明此时的 itr 即为对应的正确规则。

此外，补丁代码完善了黑洞的判断方法。原始代码有两种判断黑洞的方法：当前交换机或主机并不在网络中、当前交换机或主机在网络中但无链路与其他交换机或主机相连。

新增的判断方法是：当 itr 的值与 linkList 的尾后迭代器相等时，说明通

过 itr->rule.in_port 无法获取上一跳地址。也就是说，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找不到上一跳的交换机或者主机。

运行原始代码产生的部分错误环路如下图所示：

```
[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node
20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src
(0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), dl_dst
(1652522221583-01:80:c2:00:00:0f, 281474976710655-ff:ff:ff:ff:ff:ff), nw_src
(167772160-10.0.0.0, 167772415-10.0.0.255), nw_dst (167772160-10.0.0.0,
167772415-10.0.0.255), Field 0 (3, 3), Field 1 (0, 281474976710655), Field 2
(1652522221583, 281474976710655), Field 3 (2048, 2048), Field 4 (0, 4095), Field 5 (0,
7), Field 6 (0, 1048575), Field 7 (0, 7), Field 8 (167772160, 167772415), Field 9
(167772160, 167772415), Field 10 (0, 255), Field 11 (0, 63), Field 12 (0, 65535), Field
13 (0, 65535)
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.1 -> 20.0.0.25
```

（三）拓展实验部分

①若修改 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，试描述错误是什么，并解释出错的原因。

将 waypoint_path.py 代码中被添加规则的优先级字段改为 1，发现日志文件中无环路，但 SDC 无法 ping 通 MIT。

```
[VeriFlow::traverseForwardingGraph] The following packet class reached destination at no
[VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src (0-00:00:00:0

[VeriFlow::traverseForwardingGraph] The following packet class reached destination at no
[VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src (0-00:00:00:0

[VeriFlow::traverseForwardingGraph] The following packet class reached destination at no
[VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src (0-00:00:00:0

[VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dlSrcAddr: 00:00:00:00:00:00
[VeriFlow::verifyRule] ecCount: 3

[VeriFlow::traverseForwardingGraph] The following packet class reached destination at no
[VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src (0-00:00:00:0

[VeriFlow::traverseForwardingGraph] The following packet class reached destination at no
[VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src (0-00:00:00:0

[VeriFlow::traverseForwardingGraph] The following packet class reached destination at no
[VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src (0-00:00:00:0
```

```
mininet> SDC ping MIT
PING 10.0.0.12 (10.0.0.12) 56(84) bytes of data.
^C
--- 10.0.0.12 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3066ms
```

修改优先级前的流表：

```
test@sdnexp:~/Desktop/sdn_exp/sdn_exp_4$ sudo ovs-ofctl dump-flows s22
 cookie=0x0, duration=14.835s, table=0, n_packets=13, n_bytes=780, priority=65535,dl_dst=01:80:
c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:60
 cookie=0x0, duration=8.435s, table=0, n_packets=4, n_bytes=392, priority=1,ip,in_port="s22-eth
3",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s22-eth4"
 cookie=0x0, duration=8.430s, table=0, n_packets=5, n_bytes=490, priority=1,ip,in_port="s22-eth
4",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s22-eth3"
 cookie=0x0, duration=14.887s, table=0, n_packets=21, n_bytes=3184, priority=0 actions=CONTROLL
ER:65509
test@sdnexp:~/Desktop/sdn_exp/sdn_exp_4$ sudo ovs-ofctl dump-flows s25
 cookie=0x0, duration=16.178s, table=0, n_packets=12, n_bytes=720, priority=65535,dl_dst=01:80:
c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:60
 cookie=0x0, duration=9.997s, table=0, n_packets=4, n_bytes=392, priority=1,ip,in_port="s25-eth
2",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s25-eth1"
 cookie=0x0, duration=9.993s, table=0, n_packets=5, n_bytes=490, priority=1,ip,in_port="s25-eth
1",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s25-eth2"
 cookie=0x0, duration=16.192s, table=0, n_packets=22, n_bytes=3068, priority=0 actions=CONTROLL
ER:65509
```

修改优先级后的流表：

```
test@sdnexp:~/Desktop/sdn_exp/sdn_exp_4$ sudo ovs-ofctl dump-flows s22
 cookie=0x0, duration=217.841s, table=0, n_packets=162, n_bytes=9720, priority=65535,dl_dst=01:
80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:60
 cookie=0x0, duration=211.920s, table=0, n_packets=6, n_bytes=588, priority=1,ip,in_port="s22-e
th3",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s22-eth4"
 cookie=0x0, duration=203.401s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s22-eth
4",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s22-eth2"
 cookie=0x0, duration=203.390s, table=0, n_packets=3690, n_bytes=361620, priority=1,ip,in_port=
"s22-eth2",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s22-eth4"
 cookie=0x0, duration=217.881s, table=0, n_packets=42, n_bytes=5854, priority=0 actions=CONTROL
LER:65509
test@sdnexp:~/Desktop/sdn_exp/sdn_exp_4$ sudo ovs-ofctl dump-flows s25
 cookie=0x0, duration=242.575s, table=0, n_packets=179, n_bytes=10740, priority=65535,dl_dst=01
:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:60
 cookie=0x0, duration=236.921s, table=0, n_packets=3, n_bytes=294, priority=1,ip,in_port="s25-e
th1",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s25-eth2"
 cookie=0x0, duration=228.364s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s25-eth
3",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s25-eth2"
 cookie=0x0, duration=228.364s, table=0, n_packets=4046, n_bytes=396508, priority=1,ip,in_port=
"s25-eth2",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s25-eth3"
 cookie=0x0, duration=242.592s, table=0, n_packets=41, n_bytes=5462, priority=0 actions=CONTROL
LER:65509
```

无法 ping 通的原因：

新的流表项下发到交换机后，由于匹配字段和优先级均与先前的流表项相同，新流表项将会覆盖旧流表项，因此产生了路由环路。

无法发现环路的原因：

VeriFlow 在检测转发图中是否存在环路时，先对规则按照优先级字段进行排序：graph->links[currentLocation].sort(compareForwardingLink)，由于新下发的流表项优先级与之前的流表项相同，因此仍选择了之前的流表项进行匹配并获取下一跳地址，此时的地址就是之前的最短路径地址，故无法发现环路。

上面是我最初对于无法发现环路原因的猜测，但是当我尝试打印 graph->links[currentLocation]，却发现交换机 s22 中仅有 3 条规则，但控制器对 s22 共下发了 4 条规则。

代码如下：

```
const list< ForwardingLink >& linkList = graph->links[currentLocation];
list< ForwardingLink >::const_iterator itr = linkList.begin();

fprintf(fp, "\n");
fprintf(fp, "%s has link:\n", currentLocation.c_str());
while (itr != linkList.end()) {
    fprintf(fp, "%s\n", itr->rule.toString().c_str());
    itr++;
}
itr = linkList.begin();
```

日志文件截图如下：

```
20.0.0.22 has link:
[Rule] type: 1, dlSrcAddr: 00:00:00:00:00:00, dlSrcAddrMask: 0:0:0:0:0:0, dlDstAddr:
00:00:00:00:00:00, dlDstAddrMask: 0:0:0:0:0:0, nwSrcAddr: 10.0.0.0, nwSrcAddrMask:
255.255.255.0, nwDstAddr: 10.0.0.0, nwDstAddrMask: 255.255.255.0, location: 20.0.0.22,
nextHop: 20.0.0.23, in_port: 2, priority: 1, wildcards: 3279086, in_port: 0, dl_type:
2048, dl_vlan: 0, dl_vlan_pcp: 0, mpls_label: 0, mpls_tc: 0, nw_proto: 0, nw_tos: 0,
tp_src: 0, tp_dst: 0
[Rule] type: 1, dlSrcAddr: 00:00:00:00:00:00, dlSrcAddrMask: 0:0:0:0:0:0, dlDstAddr:
00:00:00:00:00:00, dlDstAddrMask: 0:0:0:0:0:0, nwSrcAddr: 10.0.0.0, nwSrcAddrMask:
255.255.255.0, nwDstAddr: 10.0.0.0, nwDstAddrMask: 255.255.255.0, location: 20.0.0.22,
nextHop: 20.0.0.23, in_port: 3, priority: 1, wildcards: 3279086, in_port: 0, dl_type:
2048, dl_vlan: 0, dl_vlan_pcp: 0, mpls_label: 0, mpls_tc: 0, nw_proto: 0, nw_tos: 0,
tp_src: 0, tp_dst: 0
[Rule] type: 1, dlSrcAddr: 00:00:00:00:00:00, dlSrcAddrMask: 0:0:0:0:0:0, dlDstAddr:
00:00:00:00:00:00, dlDstAddrMask: 0:0:0:0:0:0, nwSrcAddr: 10.0.0.0, nwSrcAddrMask:
255.255.255.0, nwDstAddr: 10.0.0.0, nwDstAddrMask: 255.255.255.0, location: 20.0.0.22,
nextHop: 20.0.0.15, in_port: 4, priority: 1, wildcards: 3279086, in_port: 0, dl_type:
2048, dl_vlan: 0, dl_vlan_pcp: 0, mpls_label: 0, mpls_tc: 0, nw_proto: 0, nw_tos: 0,
tp_src: 0, tp_dst: 0
```

从上图中可以发现，交换机 s22 中只有 1 条 in_port=4 的规则，但实际上应该有 2 条这样的规则。

综上所述，VeriFlow 无法发现环路的真正原因是：新流表项的匹配域和优先级均与先前的流表项相同，因此 VeriFlow 没有将这条新规则存储在 graph->links[currentLocation]中。故在交换机 s22 中，当 in_port=4 时，out_port 还是 3 而非新流表项的 2。同理，对于交换机 s25 的 in_port=2 来说，out_port 还是 1 而非新流表项的 3。故转发路径还是之前的最短路径，从而也就没有检测出环路。

②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。

VeriFlow 支持的域如下：

```
enum FieldIndex
```

```
{
    IN_PORT, // 0
    DL_SRC,
    DL_DST,
    DL_TYPE,
    DL_VLAN,
    DL_VLAN_PCP,
    MPLS_LABEL,
    MPLS_TC,
    NW_SRC,
    NW_DST,
    NW_PROTO,
    NW_TOS,
    TP_SRC,
    TP_DST,
    ALL_FIELD_INDEX_END_MARKER, // 14
    METADATA, // 15, not used in this version.
    WILDCARDS // 16
};
```

选择验证的域为：DL_SRC、DL_DST、DL_TYPE、NW_SRC、NW_DST、IN_PORT。

修改 waypoint_path.py，向下发的流表项中增加如上匹配域：

```python
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'},
data=json.dumps(flow))
    print(ret)

def show_path(src, dst, port_path):
    print('install mywaypoint 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)]

    mac1 = "00:00:00:00:00:01"
    mac2 = "00:00:00:00:00:02"

    # send flow mod
    for node in path:
        in_port, dpid, out_port = node
        add_flow(dpid, '10.0.0.0/24', '10.0.0.0/24', in_port,
out_port, mac1, mac2)
        add_flow(dpid, '10.0.0.0/24', '10.0.0.0/24', out_port,
in_port, mac2, mac1)
    show_path(src_sw, dst_sw, path)

if __name__ == '__main__':
    install_path()
```

日志文件截图如下：

```
[VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dlSrcAddr:
00:00:00:00:00:02, dlSrcAddrMask: FF:FF:FF:FF:FF:FF, dlDstAddr: 00:00:00:00:00:01,
dlDstAddrMask: FF:FF:FF:FF:FF:FF, nwSrcAddr: 10.0.0.0, nwSrcAddrMask: 255.255.255.0,
nwDstAddr: 10.0.0.0, nwDstAddrMask: 255.255.255.0, location: 20.0.0.25, nextHop:
20.0.0.7, in_port: 2, priority: 10, wildcards: 3279074, in_port: 0, dl_type: 2048,
dl_vlan: 0, dl_vlan_pcp: 0, mpls_label: 0, mpls_tc: 0, nw_proto: 0, nw_tos: 0, tp_src:
0, tp_dst: 0
[VeriFlow::verifyRule] ecCount: 1

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node
20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src
(2-00:00:00:00:00:02, 2-00:00:00:00:00:02), dl_dst (1-00:00:00:00:00:01,
1-00:00:00:00:00:01), nw_src (167772160-10.0.0.0, 167772415-10.0.0.255), nw_dst
(167772160-10.0.0.0, 167772415-10.0.0.255), Field 0 (0, 65535), Field 1 (2, 2), Field 2
(1, 1), Field 3 (2048, 2048), Field 4 (0, 4095), Field 5 (0, 7), Field 6 (0, 1048575),
Field 7 (0, 7), Field 8 (167772160, 167772415), Field 9 (167772160, 167772415), Field
10 (0, 255), Field 11 (0, 63), Field 12 (0, 65535), Field 13 (0, 65535)
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 ->
20.0.0.25
```

由于网络拓扑结构不变，且新下发的转发路径也不变，改变的仅是在匹配域中新增了源 MAC 地址与目的 MAC 地址，因此，对于特定的 MAC 地址，将会产生与原先相同的路由环路。