

SDN Experiment 4

计算机75 姚杰 2174311698

Introduction

In this experiment a new method of examining errors in the software-defined network has been introduced, as known as VeriFlow, with which we are able to analyze and tackle with the problems in the network. Therefore, this report can be roughly divided into five parts. Part 1 shall be elementary tasks and Part 2 shall be the optional task.

Environment

Operating System: Linux version 4.15.0-20-generic

RYU Controller: 4.30 version

Mininet: 2.3.0d4 version

Method

Part 1:

Printing EC counts

Let's refer to /BEADS/veriflow/VeriFlow/VeriFlow.cpp. There is a function named verifyRule, in which the VeriFlow will construct equivalence class and verify the rules that may have an impact on them. In the line 1021, uncomment the fprintf and the program will print EC counts, which is the selected line in the picture below.

```

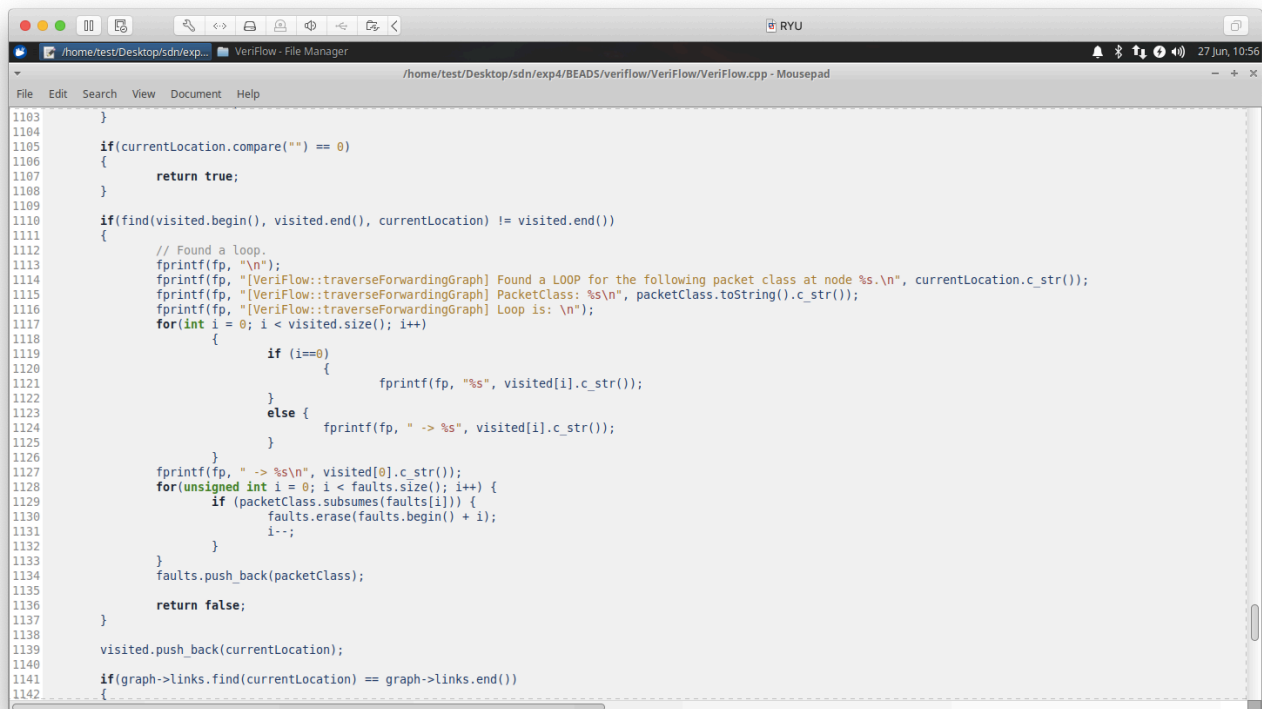
1002         return false;
1003     }
1004     gettimeofday(&end, NULL);
1005
1006     seconds = end.tv_sec - start.tv_sec;
1007     useconds = end.tv_usec - start.tv_usec;
1008     usecTime = (seconds * 1000000) + useconds;
1009     packetClassSearchTime = usecTime;
1010
1011     ecCount = vFinalPacketClasses.size();
1012     if(ecCount == 0)
1013     {
1014         fprintf(stderr, "[VeriFlow::verifyRule] Error in rule: %s\n", rule.toString().c_str());
1015         fprintf(stderr, "[VeriFlow::verifyRule] Error: (ecCount = vFinalPacketClasses.size(
1016         exit(1);
1017     }
1018     else
1019     {
1020         // fprintf(stdout, "\n");
1021         fprintf(stdout, "[VeriFlow::verifyRule] ecCount: %lu\n", ecCount);
1022     }
1023
1024     // fprintf(stdout, "[VeriFlow::verifyRule] Generating forwarding graphs...\n");
1025     gettimeofday(&start, NULL);

```

Printing the information of a loop

You may also refer to /BEADS/veriflow/VeriFlow/VeriFlow.cpp, in which there is a function named `traverseForwardGraph`. This function is used to handle all the EC graph in order to verify whether there will be loops, black holes or not. In this function, vector `<string> visited` serves as a container to collect all the nodes this function "visited". Also, this function is a recursive function, which means it will recursively traverse all nodes in the graph. In such case, the variable current location will change accordingly. Whenever the index of the current location is not the index of the last element in vector `visited`, it means current location had been visited before, indicating there is a loop in the graph, and thus the function will return a false value.

Therefore, everytime the function reports a loop, we traverse `visited` all show all the elements of it.

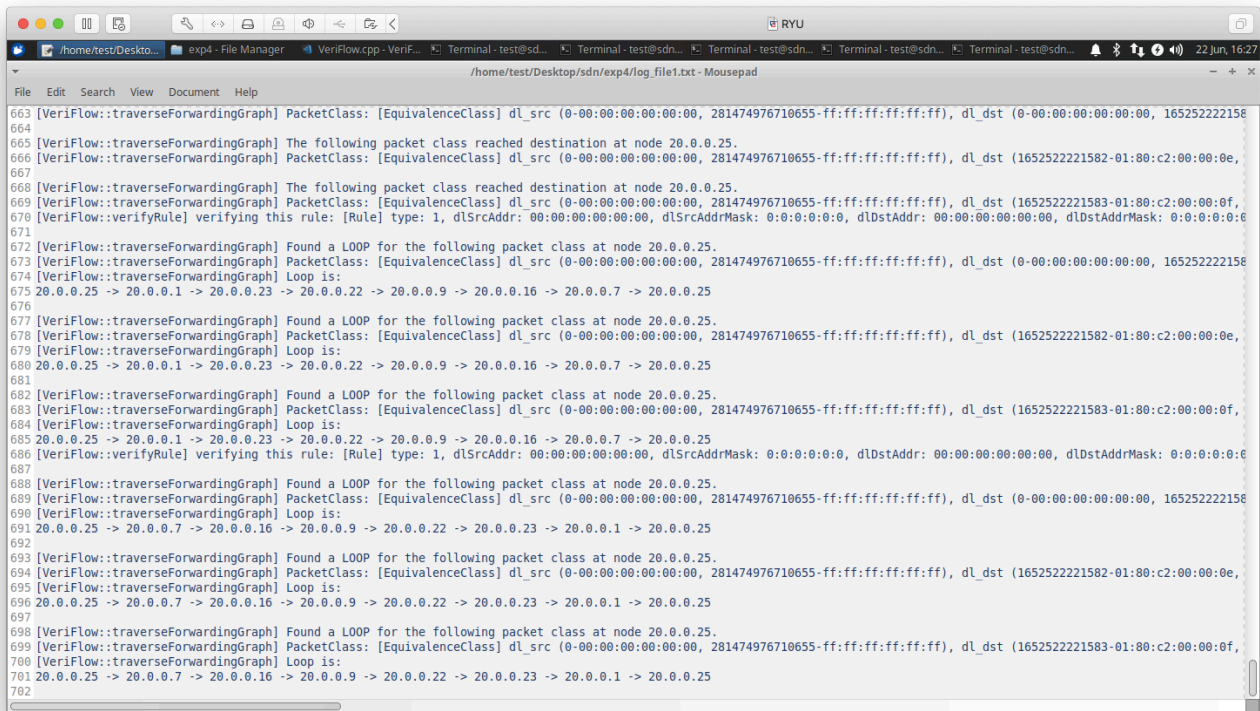


```

1103     }
1104
1105     if(currentLocation.compare("") == 0)
1106     {
1107         return true;
1108     }
1109
1110     if(find(visited.begin(), visited.end(), currentLocation) != visited.end())
1111     {
1112         // Found a loop.
1113         fprintf(fp, "\n");
1114         fprintf(fp, "[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node %s.\n", currentLocation.c_str());
1115         fprintf(fp, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n", packetClass.toString().c_str());
1116         fprintf(fp, "[VeriFlow::traverseForwardingGraph] Loop is: \n");
1117         for(int i = 0; i < visited.size(); i++)
1118         {
1119             if (i==0)
1120             {
1121                 fprintf(fp, "%s", visited[i].c_str());
1122             }
1123             else {
1124                 fprintf(fp, " -> %s", visited[i].c_str());
1125             }
1126         }
1127         fprintf(fp, "\n");
1128         for(unsigned int i = 0; i < faults.size(); i++) {
1129             if (packetClass.subsumes(faults[i])) {
1130                 faults.erase(faults.begin() + i);
1131                 i--;
1132             }
1133         }
1134         faults.push_back(packetClass);
1135         return false;
1136     }
1137
1138     visited.push_back(currentLocation);
1139
1140     if(graph->links.find(currentLocation) == graph->links.end())
1141     {
1142

```

The result shall be as follow:



```
663 [VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] d_src (0-00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), d_dst (0-00:00:00:00:00:00, 165252221582-01:80:c2:00:00:0e,
664
665 [VeriFlow::traverseForwardingGraph] The following packet class reached destination at node 20.0.0.25.
666 [VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] d_src (0-00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), d_dst (165252221582-01:80:c2:00:00:0e,
667
668 [VeriFlow::traverseForwardingGraph] The following packet class reached destination at node 20.0.0.25.
669 [VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] d_src (0-00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), d_dst (165252221582-01:80:c2:00:00:0f,
670 [VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dSrcAddr: 00:00:00:00:00, dSrcAddrMask: 0:0:0:0:0:0, dDstAddr: 00:00:00:00:00, dDstAddrMask: 0:0:0:0:0:0
671
672 [VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
673 [VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] d_src (0-00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), d_dst (0-00:00:00:00:00:00, 165252221582-01:80:c2:00:00:0e,
674 [VeriFlow::traverseForwardingGraph] Loop is:
675 20.0.0.25 -> 20.0.0.1 -> 20.0.0.23 -> 20.0.0.22 -> 20.0.0.9 -> 20.0.0.16 -> 20.0.0.7 -> 20.0.0.25
676
677 [VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
678 [VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] d_src (0-00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), d_dst (165252221582-01:80:c2:00:00:0e,
679 [VeriFlow::traverseForwardingGraph] Loop is:
680 20.0.0.25 -> 20.0.0.1 -> 20.0.0.23 -> 20.0.0.22 -> 20.0.0.9 -> 20.0.0.16 -> 20.0.0.7 -> 20.0.0.25
681
682 [VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
683 [VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] d_src (0-00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), d_dst (165252221582-01:80:c2:00:00:0f,
684 [VeriFlow::traverseForwardingGraph] Loop is:
685 20.0.0.25 -> 20.0.0.1 -> 20.0.0.23 -> 20.0.0.22 -> 20.0.0.9 -> 20.0.0.16 -> 20.0.0.7 -> 20.0.0.25
686 [VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dSrcAddr: 00:00:00:00:00, dSrcAddrMask: 0:0:0:0:0:0, dDstAddr: 00:00:00:00:00, dDstAddrMask: 0:0:0:0:0:0
687
688 [VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
689 [VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] d_src (0-00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), d_dst (0-00:00:00:00:00:00, 165252221582-01:80:c2:00:00:0e,
690 [VeriFlow::traverseForwardingGraph] Loop is:
691 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
692
693 [VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
694 [VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] d_src (0-00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), d_dst (165252221582-01:80:c2:00:00:0e,
695 [VeriFlow::traverseForwardingGraph] Loop is:
696 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
697
698 [VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
699 [VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] d_src (0-00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), d_dst (165252221582-01:80:c2:00:00:0f,
700 [VeriFlow::traverseForwardingGraph] Loop is:
701 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
702
```

Printing the simplified information of EC

```
1109
1110     if(find(visited.begin(), visited.end(), currentLocation) != visited.end())
1111     {
1112         // Found a loop.
1113         fprintf(fp, "\n");
1114         fprintf(fp, "[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node %s.\n", currentLocation.c_str());
1115         fprintf(fp, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n", packetClass.toString().c_str());
1116         fprintf(fp, "[VeriFlow::traverseForwardingGraph] Loop is: \n");
1117         for(int i = 0; i < visited.size(); i++)
1118         {
1119             if (i==0)
1120             {
1121                 fprintf(fp, "%s", visited[i].c_str());
1122             }
1123             else {
1124                 fprintf(fp, " -> %s", visited[i].c_str());
1125             }
1126         }
1127     }
```

From the fprintf function shown above, we are able to reveal that the obtaining of internal information of EC is by an internal member function called toString() of EC. (If you are using VS code or other editor, you may press CTRL and click it, then its original definition will be displayed) This function is defined in the /BEADS/veriflow/VeriFlow/EquivalenceClass.cpp, at round line 110.

Change the definition as below and you will simplify the information displayed on the screen. I'll further my explanation below.

```

109
110 string EquivalenceClass::toString() const
111 {
112     char buffer[1024];
113     /*sprintf(buffer, "[EquivalenceClass] dl_src (%lu-%s, %lu-%s), dl_dst (%lu-%s, %lu-%s)",
114                 this->lowerBound[DL_SRC], ::getMacValueAsString(this->lowerBound[DL_SRC]).c_str(),
115                 this->upperBound[DL_SRC], ::getMacValueAsString(this->upperBound[DL_SRC]).c_str(),
116                 this->lowerBound[DL_DST], ::getMacValueAsString(this->lowerBound[DL_DST]).c_str(),
117                 this->upperBound[DL_DST], ::getMacValueAsString(this->upperBound[DL_DST]).c_str());
118     */
119     //retVal += ", ";
120
121     sprintf(buffer, "nw_src (%s, %s), nw_dst (%s, %s)",
122             ::getIpValueAsString(this->lowerBound[NW_SRC]).c_str(),
123             ::getIpValueAsString(this->upperBound[NW_SRC]).c_str(),
124             //this->lowerBound[NW_DST],
125             ::getIpValueAsString(this->lowerBound[NW_DST]).c_str(),
126             //this->upperBound[NW_DST],
127             ::getIpValueAsString(this->upperBound[NW_DST]).c_str()
128             );
129
130     string retVal = buffer;
131
132     retVal += ", ";
133     sprintf(buffer, "nw_proto (%lu-%lu), tp_src(%lu-%lu), tp_dst(%lu-%lu) ",
134             this->lowerBound[10],
135             this->upperBound[10],
136             this->lowerBound[12],
137             this->upperBound[12],
138             this->lowerBound[13],
139             this->upperBound[13]);
140
141     retVal += buffer;
142

```

As you shall see, the variable retVal is used to record all the information you concerned and it will be returned. The array lowerBound and upperBound are both unsigned long int arrays having details of an EC. The index can be gained in the /BEADS/veriflow/VeriFlow/EquivalenceClass.h. Since we only need to print 5 domains, just find their names or index numbers.

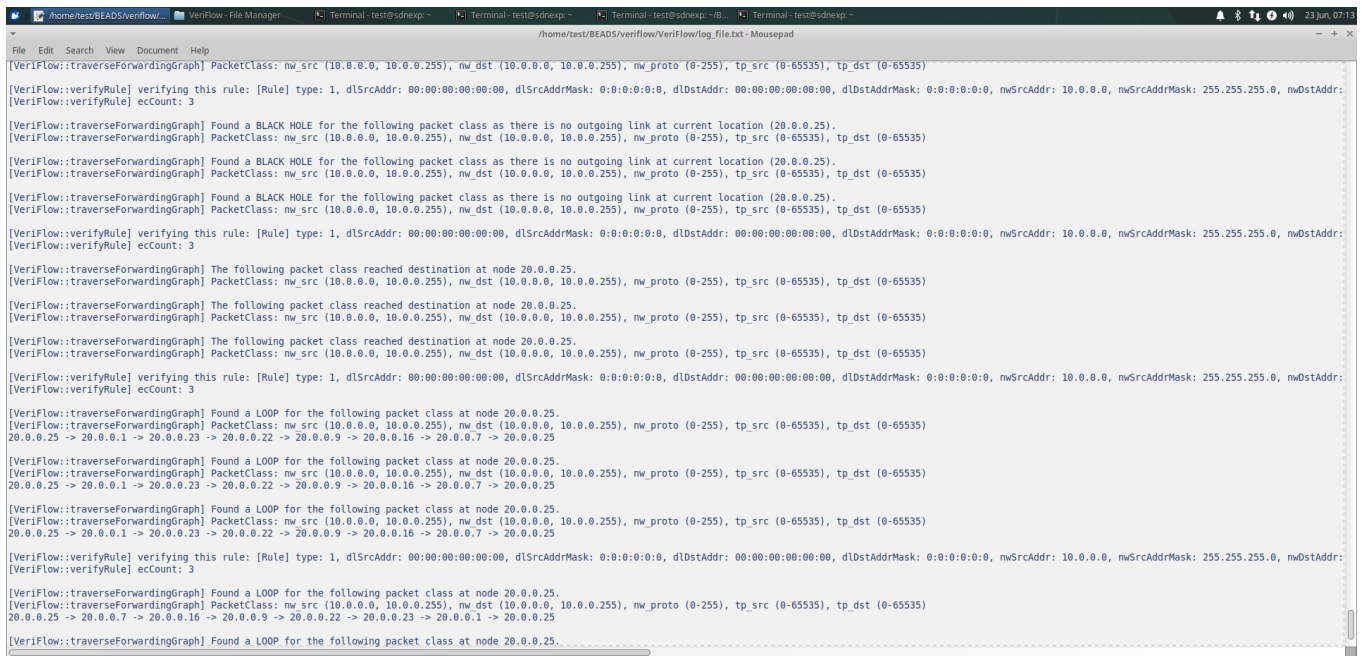
```

19 #include <sys/types.h>
20 #include <unistd.h>
21 #include <stdint.h>
22 #include <string>
23
24 using namespace std;
25
26 enum FieldIndex
27 {
28
29     IN_PORT, // 0
30     DL_SRC, // 1
31     DL_DST, // 2
32     DL_TYPE, // 3
33     DL_VLAN, // 4
34     DL_VLAN_PCP, // 5
35     MPLS_LABEL, // 6
36     MPLS_TC, // 7|
37
38     NW_SRC, // 8
39     NW_DST, // 9
40     NW_PROTO, // 10
41     NW_TOS, // 11
42     TP_SRC, // 12
43     TP_DST, // 13
44     ALL_FIELD_INDEX_END_MARKER, // 14
45     METADATA, // 15, not used in this version.
46     WILDCARDS // 16
47 };
48
49 const unsigned int fieldWidth[] = {16, 48, 48, 16, 12, 3, 20, 3, 32, 32, 8, 6, 16, 16, 0, 64, 32};
50 //{32,32,8,16,16};
51 class EquivalenceClass
52 {

```

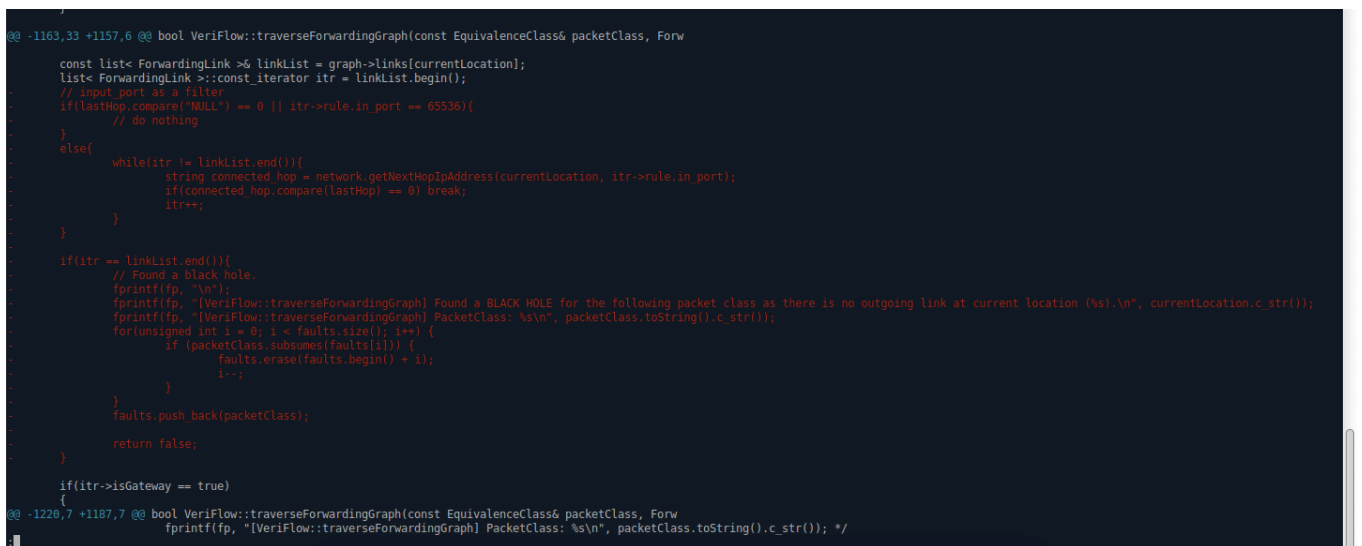
Names or index numbers can both serve as an index. For instance, both lowerBound[0] and lowerBound[in_port] work fine.

The result shall be as follow:



Understandings of the patch

There are two main changes in the patch. First, the modification of the rule class. The member attributes of rule class, `fieldValue[in_port]` and `fieldMask[in_port]` have been abandoned somehow. Instead, the member attributes, `in_port` has been adopted. Second, a new method of detecting black holes has been added.



As is shown in the screenshot above, when traverse the graph, last hop will record the actual last hop. The iterator `itr` will point to every node in the linklist. As `itr` changes the object it points to, `connected_hop` points to the last node corresponding to different `in_ports` under all rules. There must be such a `connected_hop`, from which the data packet can match the rules so that it can be forwarded at this node. If the `connected_hop` identical to the last hop (the node corresponding to the port where the data packet actually comes in) is not found in the end, it means that there are various rules at this node, but the data packet does not match any rules at this node, so a black hole is formed.

You should be alerted that the function: `getNextHopIPAddress` does not return the ip address of next hop as it is called. Inside it, there is a map which mapping an unsigned int to a string. Instead, it returns the ip address corresponding to the port regardless whether such a port is `in_port` or `out_port`.

为防止英文叙述不清楚，再用中文说一遍。

`connected_hop`指向在所有规则下面，不同in port对应的上一个节点，必须要存在这样的上一跳，数据包在这

个节点才能匹配到能转发的规则。倘若直到最后都找不到和last hop（数据包实际进来的端口对应的节点），说明在这个节点有各种转发出去的规则，但是数据包在这个节点匹配不上任何规则，因此其无法被转发，就形成了黑洞。

Moreover, these 3 ways of detecting black holes in the function `traverseForwardingGraph()` can be concluded as follows:

1. Current location (or node) does not exist in the graph.
2. There is no outgoing link (or rule) in the current location.
3. There is no outgoing match for the last hop in current location even if there are outgoing rules here.

Part 2: Optional Tasks

In the previous part we have tackled with basic tasks including several ways of modifying functions in an attempt to print desired information.

In this part we change the priority of flows distributed by the controller to the switches, from 10 to 1. Afterwards, it can be observed that SDC ping MIT won't work yet VeriFlow will still judge that the data packet can be sent from SDC ping MIT.

```
*** Starting CLI:
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=2 ttl=64 time=197 ms
64 bytes from 10.0.0.12: icmp_seq=3 ttl=64 time=235 ms
64 bytes from 10.0.0.12: icmp_seq=4 ttl=64 time=234 ms
64 bytes from 10.0.0.12: icmp_seq=5 ttl=64 time=233 ms
64 bytes from 10.0.0.12: icmp_seq=6 ttl=64 time=234 ms
64 bytes from 10.0.0.12: icmp_seq=7 ttl=64 time=233 ms
^C
--- 10.0.0.12 ping statistics ---
7 packets transmitted, 6 received, 14% packet loss, time 6035ms
rtt min/avg/max/mdev = 197.365/228.028/235.598/13.735 ms
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 ---
15 packets transmitted, 0 received, 100% packet loss, time 14335ms

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 ---
6 packets transmitted, 0 received, 100% packet loss, time 5111ms
```

First, let me explain why SDC cannot ping MIT. We use

```
1 dpctl dump-flows
```

to print all the flows on the switches. The result is:


```
*** s11
cookie=>0, duration=241.417s, table=0, n_packets=120, n_bytes=7200, priority=65535,d1_dst=01:80:c2:00:00:0e,d1_type=0x8Bcc actions=CONTROLLER:60
cookie=>0, duration=241.401s, table=0, n_packets=10, n_bytes=1550, priority=0 actions=CONTROLLER:65509
*** s12
cookie=>0, duration=241.423s, table=0, n_packets=121, n_bytes=7260, priority=65535,d1_dst=01:80:c2:00:00:0e,d1_type=0x8Bcc actions=CONTROLLER:60
cookie=>0, duration=241.407s, table=0, n_packets=26, n_bytes=2264, priority=0 actions=CONTROLLER:65509
*** s13
cookie=>0, duration=241.476s, table=0, n_packets=183, n_bytes=10800, priority=65535,d1_dst=01:80:c2:00:00:0e,d1_type=0x8Bcc actions=CONTROLLER:60
cookie=>0, duration=241.433s, table=0, n_packets=26, n_bytes=2264, priority=0 actions=CONTROLLER:65509
*** s14
cookie=>0, duration=241.426s, table=0, n_packets=121, n_bytes=7260, priority=65535,d1_dst=01:80:c2:00:00:0e,d1_type=0x8Bcc actions=CONTROLLER:60
cookie=>0, duration=241.478s, table=0, n_packets=20, n_bytes=1690, priority=0 actions=CONTROLLER:65509
*** s15
cookie=>0, duration=241.481s, table=0, n_packets=122, n_bytes=7320, priority=65535,d1_dst=01:80:c2:00:00:0e,d1_type=0x8Bcc actions=CONTROLLER:60
cookie=>0, duration=239.728s, table=0, n_packets=27, n_bytes=2640, priority=1,i.p.in port="s15-eth1",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s15-eth3"
cookie=>0, duration=239.682s, table=0, n_packets=6, n_bytes=588, priority=1,i.p.in port="s15-eth3",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s15-eth1"
*** s16
cookie=>0, duration=241.438s, table=0, n_packets=23, n_bytes=1822, priority=0 actions=CONTROLLER:65509
*** s17
cookie=>0, duration=239.452s, table=0, n_packets=177, n_bytes=10620, priority=65535,d1_dst=01:80:c2:00:00:0e,d1_type=0x8Bcc actions=CONTROLLER:60
cookie=>0, duration=172.710s, table=0, n_packets=0, n_bytes=0, priority=1,i.p.in port="s16-eth3",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s16-eth2"
cookie=>0, duration=172.707s, table=0, n_packets=17184, n_bytes=107032, priority=1,i.p.in port="s15-eth2",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s16-eth3"
cookie=>0, duration=239.494s, table=0, n_packets=27, n_bytes=2234, priority=0 actions=CONTROLLER:65509
*** s17
cookie=>0, duration=241.439s, table=0, n_packets=119, n_bytes=7140, priority=65535,d1_dst=01:80:c2:00:00:0e,d1_type=0x8Bcc actions=CONTROLLER:60
cookie=>0, duration=241.483s, table=0, n_packets=19, n_bytes=1626, priority=0 actions=CONTROLLER:65509
*** s18
cookie=>0, duration=241.442s, table=0, n_packets=182, n_bytes=10920, priority=65535,d1_dst=01:80:c2:00:00:0e,d1_type=0x8Bcc actions=CONTROLLER:60
cookie=>0, duration=241.483s, table=0, n_packets=26, n_bytes=2264, priority=0 actions=CONTROLLER:65509
*** s19
cookie=>0, duration=241.449s, table=0, n_packets=120, n_bytes=7200, priority=65535,d1_dst=01:80:c2:00:00:0e,d1_type=0x8Bcc actions=CONTROLLER:60
cookie=>0, duration=241.491s, table=0, n_packets=17, n_bytes=1486, priority=0 actions=CONTROLLER:65509
*** s20
cookie=>0, duration=241.451s, table=0, n_packets=122, n_bytes=7320, priority=65535,d1_dst=01:80:c2:00:00:0e,d1_type=0x8Bcc actions=CONTROLLER:60
cookie=>0, duration=241.493s, table=0, n_packets=19, n_bytes=1626, priority=0 actions=CONTROLLER:65509
*** s21
cookie=>0, duration=241.451s, table=0, n_packets=121, n_bytes=7260, priority=65535,d1_dst=01:80:c2:00:00:0e,d1_type=0x8Bcc actions=CONTROLLER:60
cookie=>0, duration=241.493s, table=0, n_packets=18, n_bytes=1556, priority=0 actions=CONTROLLER:65509
*** s22
cookie=>0, duration=241.454s, table=0, n_packets=181, n_bytes=10860, priority=65535,d1_dst=01:80:c2:00:00:0e,d1_type=0x8Bcc actions=CONTROLLER:60
cookie=>0, duration=239.748s, table=0, n_packets=26, n_bytes=2548, priority=1,i.p.in port="s22-eth3",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s22-eth4"
cookie=>0, duration=172.746s, table=0, n_packets=0, n_bytes=0, priority=1,i.p.in port="s22-eth4",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s22-eth2"
cookie=>0, duration=172.742s, table=0, n_packets=17005, n_bytes=1674330, priority=1,i.p.in port="s22-eth2",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s22-eth4"
cookie=>0, duration=241.497s, table=0, n_packets=27, n_bytes=2234, priority=0 actions=CONTROLLER:65509
*** s23
cookie=>0, duration=241.482s, table=0, n_packets=183, n_bytes=10800, priority=65535,d1_dst=01:80:c2:00:00:0e,d1_type=0x8Bcc actions=CONTROLLER:60
cookie=>0, duration=239.747s, table=0, n_packets=17111, n_bytes=1678878, priority=1,i.p.in port="s23-eth4",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s23-eth2"
cookie=>0, duration=239.745s, table=0, n_packets=4, n_bytes=588, priority=1,i.p.in port="s23-eth2",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s23-eth4"
cookie=>0, duration=241.583s, table=0, n_packets=26, n_bytes=2264, priority=0 actions=CONTROLLER:65509
*** s24
cookie=>0, duration=241.484s, table=0, n_packets=120, n_bytes=7200, priority=65535,d1_dst=01:80:c2:00:00:0e,d1_type=0x8Bcc actions=CONTROLLER:60
cookie=>0, duration=241.509s, table=0, n_packets=19, n_bytes=1626, priority=0 actions=CONTROLLER:65509
*** s25
cookie=>0, duration=239.483s, table=0, n_packets=177, n_bytes=10620, priority=65535,d1_dst=01:80:c2:00:00:0e,d1_type=0x8Bcc actions=CONTROLLER:60
cookie=>0, duration=239.706s, table=0, n_packets=6, n_bytes=588, priority=1,i.p.in port="s25-eth1",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s25-eth2"
cookie=>0, duration=172.726s, table=0, n_packets=0, n_bytes=0, priority=1,i.p.in port="s25-eth3",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s25-eth2"
cookie=>0, duration=172.722s, table=0, n_packets=17184, n_bytes=107032, priority=1,i.p.in port="s25-eth2",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s25-eth3"
cookie=>0, duration=239.525s, table=0, n_packets=31, n_bytes=2582, priority=0 actions=CONTROLLER:65509
mininet[0]
```

Note that in the s22 and s25 some of the original flows have been covered, and some new have been added. Take s22 for example, there are 3 flows affecting packets:

in port	connected school	out port	connected school
eth3	SDC	eth4	UTAH
eth4	UTAH	eth2	Tinker
eth2	Tinker	eth4	UTAH

That makes a loop. Let SDC ping MIT. The packet is first passed through SDC, USC, UTAH, ILLINOIS and MIT. When it travels back, it will pass through MIT, ILLINOIS, UTAH and USC. In the USC, according to new rules, it has to go to TINKER. It's alike in the MIT switch. So there shall be a loop.

However, VeriFlow cannot reveal this loop.

```
File Edit Search View Document Help
/home/test/BEADS/veriflow/VeriFlow/log_file.pri1.txt - Mousepad

775
776 [VeriFlow::traverseForwardingGraph] The following packet class reached destination at node 20.0.0.25.
777 [VeriFlow::traverseForwardingGraph] PacketClass: mw_src (10.0.0.0, 10.0.0.255), mw_dst (10.0.0.0, 10.0.0.255), mw_proto (0-255), tp_src (0-65535), tp_dst (0-65535)
778
779 [VeriFlow::traverseForwardingGraph] The following packet class reached destination at node 20.0.0.25.
780 [VeriFlow::traverseForwardingGraph] PacketClass: mw_src (10.0.0.0, 10.0.0.255), mw_dst (10.0.0.0, 10.0.0.255), mw_proto (0-255), tp_src (0-65535), tp_dst (0-65535)
781
782 [VeriFlow::traverseForwardingGraph] The following packet class reached destination at node 20.0.0.25.
783 [VeriFlow::traverseForwardingGraph] PacketClass: mw_src (10.0.0.0, 10.0.0.255), mw_dst (10.0.0.0, 10.0.0.255), mw_proto (0-255), tp_src (0-65535), tp_dst (0-65535)
784
785 [VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dSrcAddr: 00:00:00:00:00:00, dSrcAddrMask: 0:0:0:0:0:0, dDstAddr: 00:00:00:00:00:00, dDstAddrMask: 0:0:0:0:0:0, mwSrcAddr: 10.0.0.0, mwSrcAddrMask: 255.255.255.0, mwDstA
786 [VeriFlow::verifyRule] eccount: 3
787
788 [VeriFlow::traverseForwardingGraph] (New Function) Found a BLACK HOLE at current location (20.0.0.25).
789 LastHop: 20.0.0.7. s
790 [VeriFlow::traverseForwardingGraph] PacketClass: mw_src (10.0.0.0, 10.0.0.255), mw_dst (10.0.0.0, 10.0.0.255), mw_proto (0-255), tp_src (0-65535), tp_dst (0-65535)
791
792 [VeriFlow::traverseForwardingGraph] (New Function) Found a BLACK HOLE at current location (20.0.0.25).
793 LastHop: 20.0.0.7. s
794 [VeriFlow::traverseForwardingGraph] PacketClass: mw_src (10.0.0.0, 10.0.0.255), mw_dst (10.0.0.0, 10.0.0.255), mw_proto (0-255), tp_src (0-65535), tp_dst (0-65535)
795
796 [VeriFlow::traverseForwardingGraph] (New Function) Found a BLACK HOLE at current location (20.0.0.25).
797 LastHop: 20.0.0.7. s
798 [VeriFlow::traverseForwardingGraph] PacketClass: mw_src (10.0.0.0, 10.0.0.255), mw_dst (10.0.0.0, 10.0.0.255), mw_proto (0-255), tp_src (0-65535), tp_dst (0-65535)
799
800 [VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dSrcAddr: 00:00:00:00:00:00, dSrcAddrMask: 0:0:0:0:0:0, dDstAddr: 00:00:00:00:00:00, dDstAddrMask: 0:0:0:0:0:0, mwSrcAddr: 10.0.0.0, mwSrcAddrMask: 255.255.255.0, mwDstA
801 [VeriFlow::verifyRule] eccount: 3
802
803 [VeriFlow::traverseForwardingGraph] The following packet class reached destination at node 20.0.0.25.
804 [VeriFlow::traverseForwardingGraph] PacketClass: mw_src (10.0.0.0, 10.0.0.255), mw_dst (10.0.0.0, 10.0.0.255), mw_proto (0-255), tp_src (0-65535), tp_dst (0-65535)
805
806 [VeriFlow::traverseForwardingGraph] The following packet class reached destination at node 20.0.0.25.
807 [VeriFlow::traverseForwardingGraph] PacketClass: mw_src (10.0.0.0, 10.0.0.255), mw_dst (10.0.0.0, 10.0.0.255), mw_proto (0-255), tp_src (0-65535), tp_dst (0-65535)
808
809 [VeriFlow::traverseForwardingGraph] The following packet class reached destination at node 20.0.0.25.
810 [VeriFlow::traverseForwardingGraph] PacketClass: mw_src (10.0.0.0, 10.0.0.255), mw_dst (10.0.0.0, 10.0.0.255), mw_proto (0-255), tp_src (0-65535), tp_dst (0-65535)
811
812 [VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dSrcAddr: 00:00:00:00:00:00, dSrcAddrMask: 0:0:0:0:0:0, dDstAddr: 00:00:00:00:00:00, dDstAddrMask: 0:0:0:0:0:0, mwSrcAddr: 10.0.0.0, mwSrcAddrMask: 255.255.255.0, mwDstA
813 [VeriFlow::verifyRule] eccount: 3
814
815 [VeriFlow::traverseForwardingGraph] The following packet class reached destination at node 20.0.0.15.
816 [VeriFlow::traverseForwardingGraph] PacketClass: mw_src (10.0.0.0, 10.0.0.255), mw_dst (10.0.0.0, 10.0.0.255), mw_proto (0-255), tp_src (0-65535), tp_dst (0-65535)
817
818 [VeriFlow::traverseForwardingGraph] The following packet class reached destination at node 20.0.0.15.
819 [VeriFlow::traverseForwardingGraph] PacketClass: mw_src (10.0.0.0, 10.0.0.255), mw_dst (10.0.0.0, 10.0.0.255), mw_proto (0-255), tp_src (0-65535), tp_dst (0-65535)
820
821 [VeriFlow::traverseForwardingGraph] The following packet class reached destination at node 20.0.0.15.
822 [VeriFlow::traverseForwardingGraph] PacketClass: mw_src (10.0.0.0, 10.0.0.255), mw_dst (10.0.0.0, 10.0.0.255), mw_proto (0-255), tp_src (0-65535), tp_dst (0-65535)
823
```

In the log file, it show thats the packet can reach 20.0.0.25 yet doesnot claim that there will be an error of loop.

That's because only the switch will cover old flows if the match domains are identical to those of the old flows. Nor will VeriFlow. Therefore, in VeriFlow the old flows will not be deleted and thus by calculation loops will not be produced.

Reference

[OpenFlow Switch Specification](#)

Source code

Since all the codes have been given explicitly in the text above, it is not necessary to give them here.