SDN Experiment 4

计算机75 姚杰 2174311698

Introduction

In this experiment a new method of examing 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

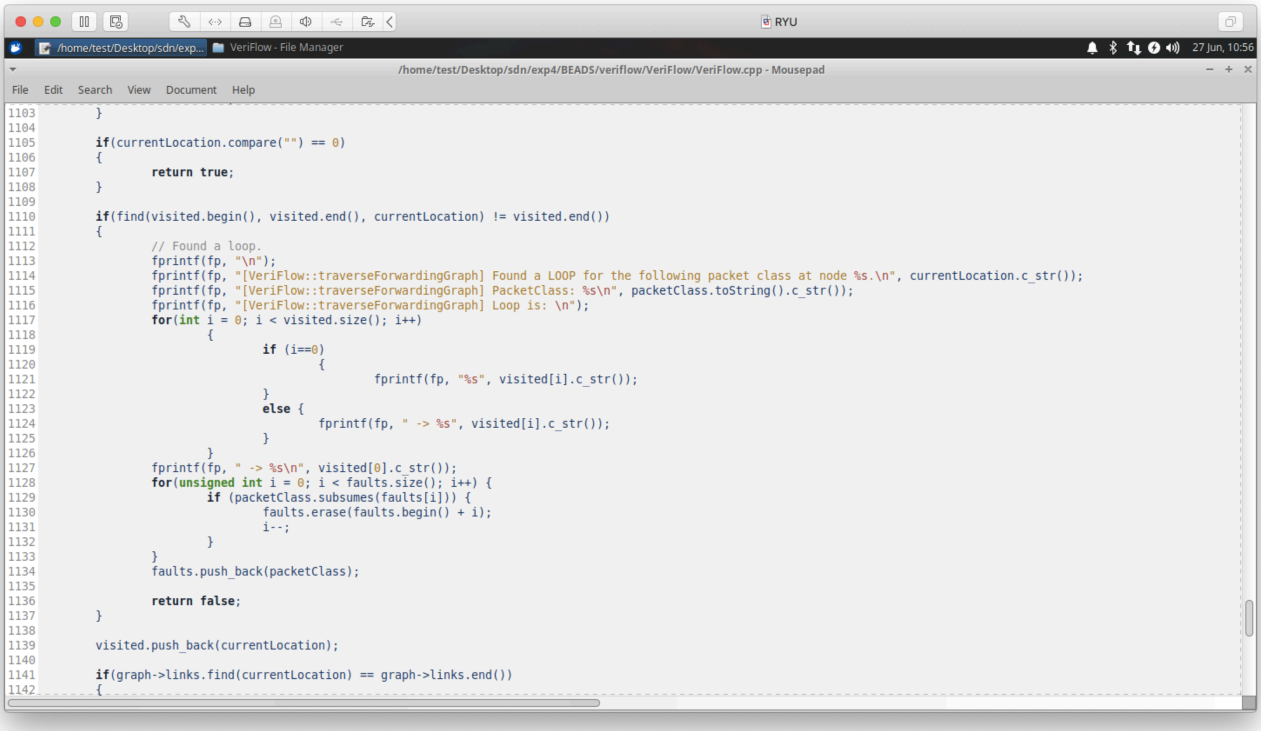
Let's refer to /BEADS/veriflow/VeriFlow/VeriFlow.cpp. There is a function named verifyRule, in which the VeriFlow will construct equivalence class and verify the rules that may have an impact on them. In the line 1021, uncomment the fprintf and the program will print EC counts, which is the selected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

```
                return false;
        }
        gettimeofday(&end, NULL);

        seconds  = end.tv_sec  - start.tv_sec;
        useconds = end.tv_usec - start.tv_usec;
        usecTime = (seconds * 1000000) + useconds;
        packetClassSearchTime = usecTime;

        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(
                exit(1);
        }
        else
        {
                // fprintf(stdout, "\n");
                fprintf(stdout, "[VeriFlow::verifyRule] ecCount: %lu\n", ecCount);
        }

        // fprintf(stdout, "[VeriFlow::verifyRule] Generating forwarding graphs...\n");
        gettimeofday(&start, NULL);
```

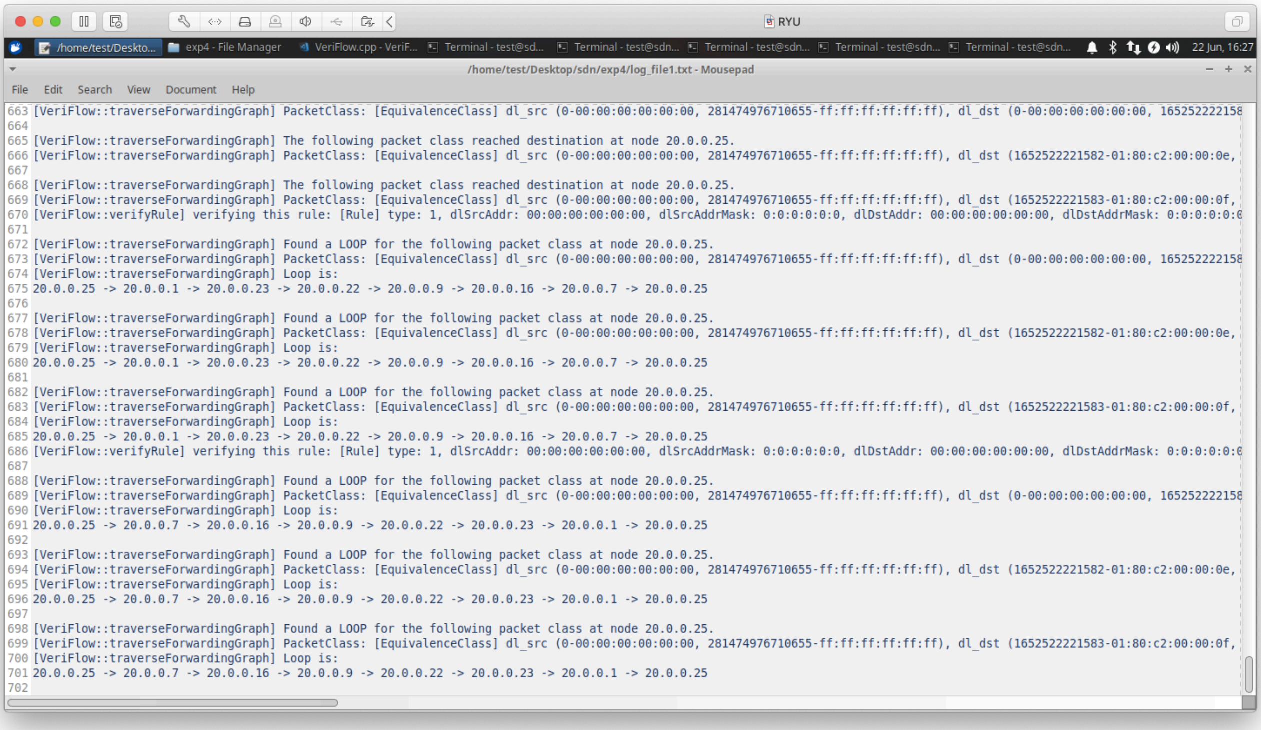
Printing the information of a loop

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.



The result shall be as follow:



```
[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, 165252222158

[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 (1652522221582-01:80:c2:00:00:0e,

[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 (1652522221583-01:80:c2:00:00:0f,
[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

[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 (0-00:00:00:00:00:00, 165252222158
[VeriFlow::traverseForwardingGraph] Loop is:
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

[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,
[VeriFlow::traverseForwardingGraph] Loop is:
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

[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,
[VeriFlow::traverseForwardingGraph] Loop is:
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
[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

[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 (0-00:00:00:00:00:00, 165252222158
[VeriFlow::traverseForwardingGraph] Loop 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

[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,
[VeriFlow::traverseForwardingGraph] Loop 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

[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,
[VeriFlow::traverseForwardingGraph] Loop 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
```

Printing the simplified information of EC

```
        if(find(visited.begin(), visited.end(), 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 is: \n");
                for(int i = 0; i < visited.size(); i++)
                        {
                                if (i==0)
                                        {
                                                fprintf(fp, "%s", visited[i].c_str());
                                }
                                else {
                                        fprintf(fp, " -> %s", visited[i].c_str());
                                }
                        }
```

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 defied 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 explaination below.

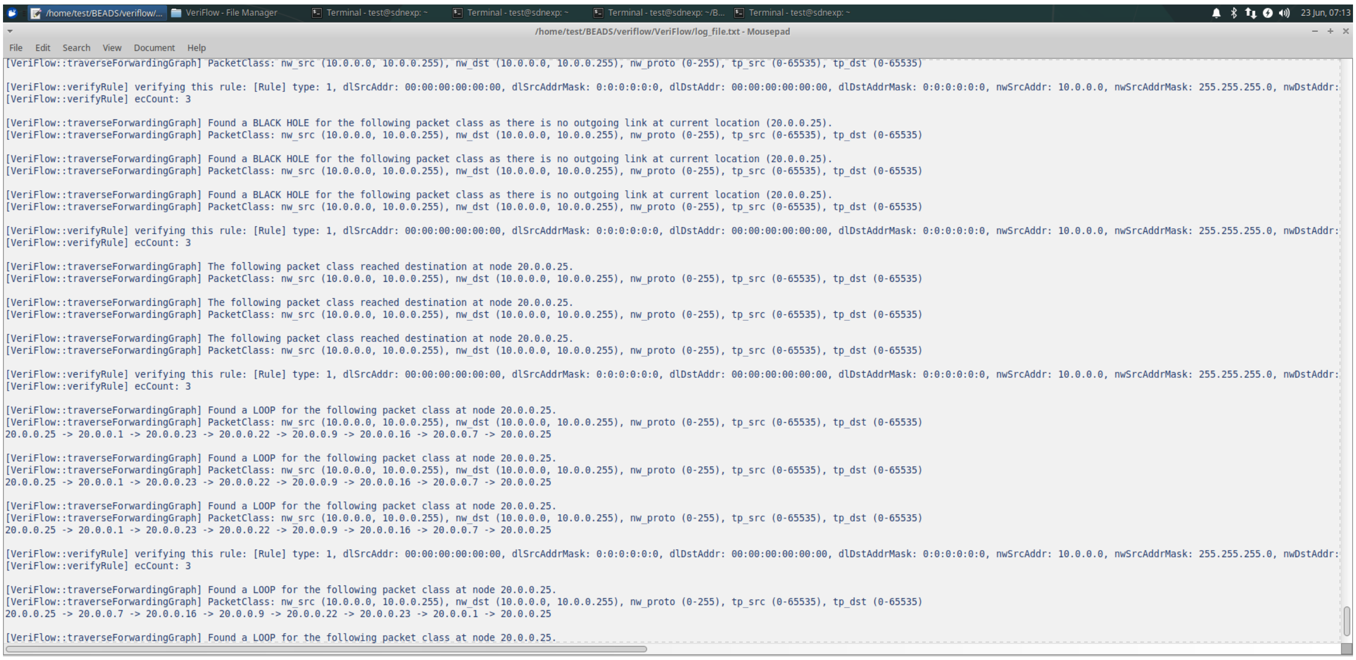
```
string EquivalenceClass::toString() const
{
	char buffer[1024];
	/*sprintf(buffer, "[EquivalenceClass] dl_src (%lu-%s, %lu-%s), dl_dst (%lu-%s, %lu-%s)",
			this->lowerBound[DL_SRC], ::getMacValueAsString(this->lowerBound[DL_SRC]).c_str(),
			this->upperBound[DL_SRC], ::getMacValueAsString(this->upperBound[DL_SRC]).c_str(),
			this->lowerBound[DL_DST], ::getMacValueAsString(this->lowerBound[DL_DST]).c_str(),
			this->upperBound[DL_DST], ::getMacValueAsString(this->upperBound[DL_DST]).c_str());
	*/
	//retVal += ", ";

	sprintf(buffer, "nw_src (%s, %s), nw_dst (%s, %s)",
			::getIpValueAsString(this->lowerBound[NW_SRC]).c_str(),
			::getIpValueAsString(this->upperBound[NW_SRC]).c_str(),
			//this->lowerBound[NW_DST],
			::getIpValueAsString(this->lowerBound[NW_DST]).c_str(),
			//this->upperBound[NW_DST],
			::getIpValueAsString(this->upperBound[NW_DST]).c_str()
			);

	string retVal = buffer;

	retVal += ", ";
	sprintf(buffer, "nw_proto (%lu-%lu), tp_src(%lu-%lu), tp_dst(%lu-%lu) ",
		this->lowerBound[10],
		this->upperBound[10],
		this->lowerBound[12],
		this->upperBound[12],
		this->lowerBound[13],
		this->upperBound[13]);

	retVal += buffer;

```

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.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdint.h>
#include <string>

using namespace std;

enum FieldIndex
{

	IN_PORT, // 0
	DL_SRC, // 1
	DL_DST, // 2
	DL_TYPE, // 3
	DL_VLAN, // 4
	DL_VLAN_PCP, // 5
	MPLS_LABEL, // 6
	MPLS_TC, // 7

	NW_SRC, // 8
	NW_DST, // 9
	NW_PROTO, // 10
	NW_TOS, // 11
	TP_SRC, // 12
	TP_DST, // 13
	ALL_FIELD_INDEX_END_MARKER, // 14
	METADATA, // 15, not used in this version.
	WILDCARDS // 16
};

const unsigned int fieldWidth[] = {16, 48, 48, 16, 12, 3, 20, 3, 32, 32, 8, 6, 16, 16, 0, 64, 32};
//{32,32,8,16,16};
class EquivalenceClass
{
```

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:



```
/home/test/BEADS/veriflow/VeriFlow/log_file.txt - Mousepad
File  Edit  Search  View  Document  Help
[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::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:
[VeriFlow::verifyRule] ecCount: 3

[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: 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] 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: 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] 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: 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::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:
[VeriFlow::verifyRule] ecCount: 3

[VeriFlow::traverseForwardingGraph] The following packet class reached destination 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] The following packet class reached destination 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] The following packet class reached destination 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::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:
[VeriFlow::verifyRule] ecCount: 3

[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)
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

[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)
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

[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)
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

[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:
[VeriFlow::verifyRule] ecCount: 3

[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)
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

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
```

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.

```
@@ -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 class as there is no outgoing link at current location (%s).\n", currentLocation.c_str());
-               fprintf(fp, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n", packetClass.toString().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;
-       }

        if(itr->isGateway == true)
        {
@@ -1220,7 +1187,7 @@ bool VeriFlow::traverseForwardingGraph(const EquivalenceClass& packetClass, Forw
                        fprintf(fp, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n", packetClass.toString().c_str()); */
:
```

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 an 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.

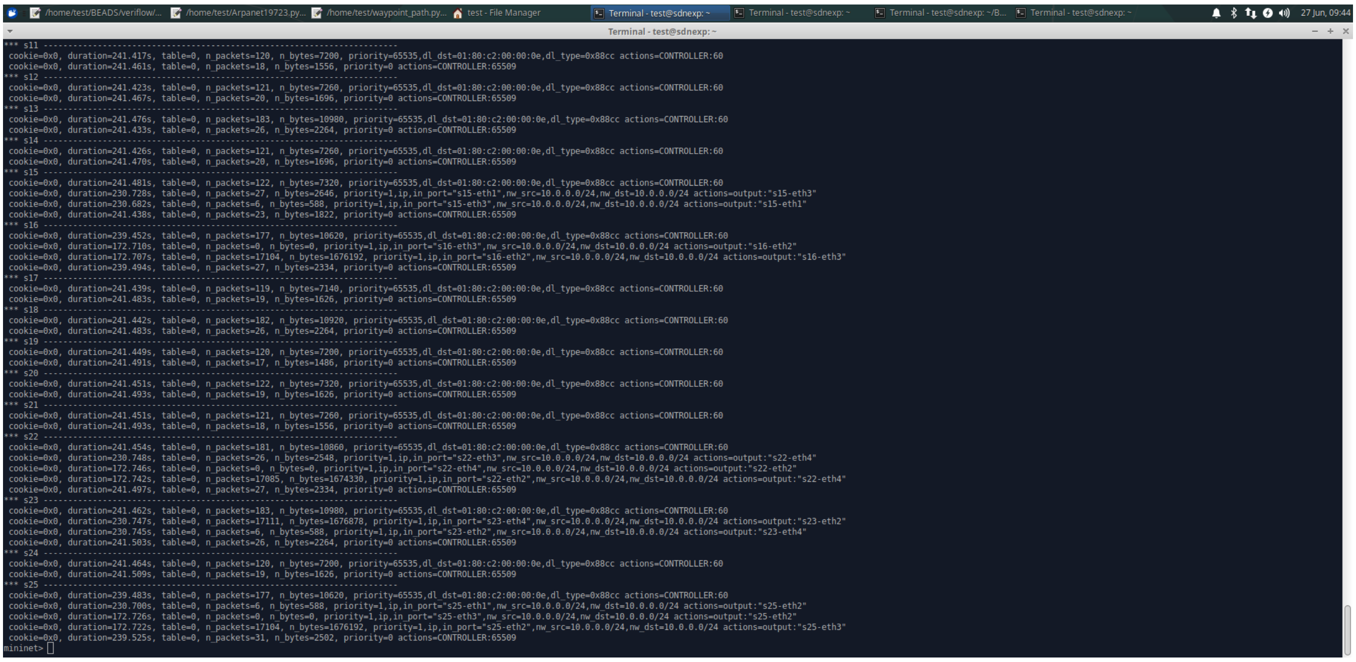
In this part we change the priority of flows distributed by the controller to the switches, from 10 to 1. Afterwards, it can be observed that SDC ping MIT won't work yet VeriFlow will still judge that the data packet can be sent from SDC ping MIT.

```
*** 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

```
dpctl dump-flows
```

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

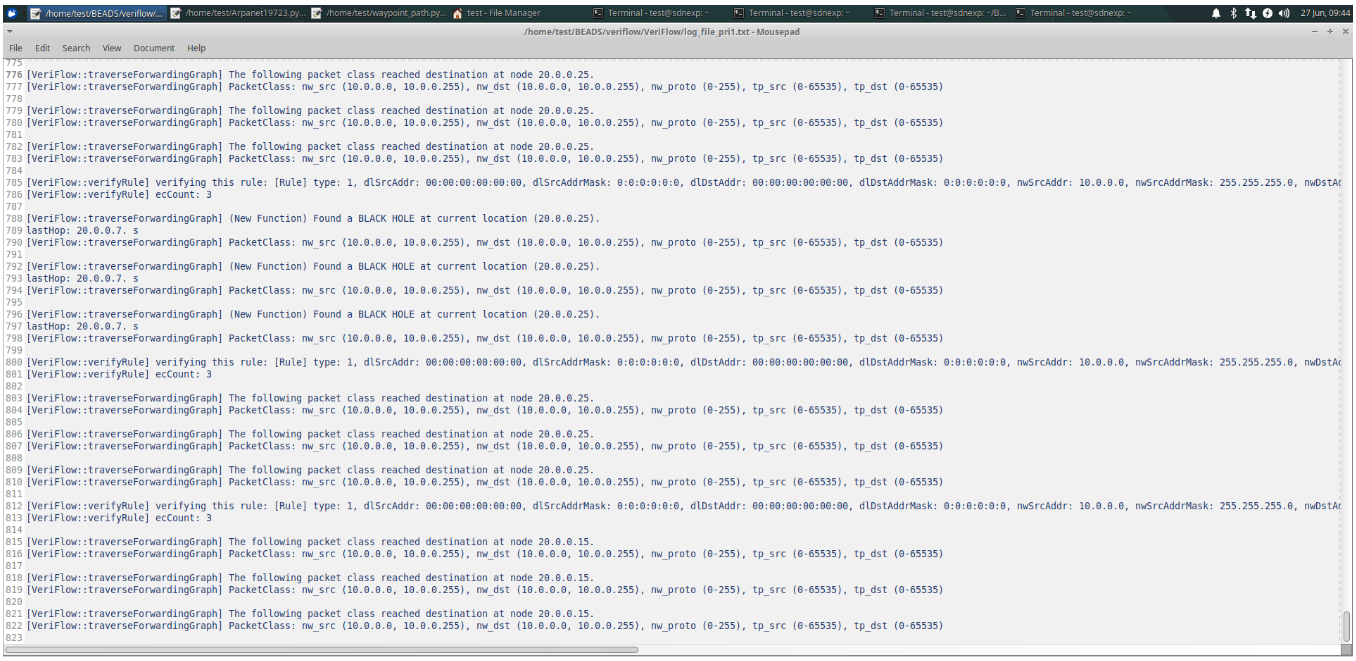


Note that in the s22 and s25 some of the original flows have been covered, and some new have been added. Take s22 for example, there are 3 flows affecting packets:

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.



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 expicitly in the text above, it is not necessary to give them here.