RTE Flow Test Suite

Why an Exhaustive Test Suite is not Practical

When I started on this test case, as I mentioned in the CI meeting, I decided to build a type system to handle this and produce all permutations of output. I intended to fuzz test the API. Once I had enough of the types implemented for patterns to test them out, and I don't think doing all the permutations out is going to work. If I:

- Restrict the depth of pattern items to 20
    - It is possible to reach an infinite depth and have a valid flow rule via no-ops, nested tunneling protocols, and backtracking via the “raw” pattern item.
- Don't allow nested tunneling
- Throw out the protocols that scapy doesn't support
- Don’t use any of the parameters for any pattern item
- Only include rules that would match sensible packets
    - Nothing where OSI levels are out of order

- Nothing where layers are omitted such as Ether / IP / HTTP

Then I get 2,549,026 patterns. Since actions seem to be able to override one another, it seems to be legal to have all actions in a single rule, with it then taking the default action of letting other rules handle it (This behavior is specified in the specification for the flow syntax). There are 52 actions, and that means I need to multiply the number of patterns by the number of valid sets of actions. This turns out to be 2^52. This gives me 11,479,792,543,757,705,936,896 possible flows using only 1 port, 1 queue, 1 group, and ignoring some of the other context options like incoming and outgoing.

Given those numbers, it is not possible to fuzz the API or even test a reasonable percentage of it. As such, measures will need to be taken to reduce the number of rules tested. In general, these will involve testing rules which are likely to be used, as well as rules that, while less common in normal networking, are the types of rules that an application likely to make use of DPDK would need (ex; a DNS server, an HTTP server, a firewall, etc). Suggestions regarding rules which should be included are welcome.

Pattern Matching

This test case will consist of creating some packets representative of common network traffic patterns and then creating flow rules using the COUNT action to validate that these patterns function correctly. This representation of common network traffic patterns will consist mainly of non tunneled TCP and UDP traffic. The largest focuses will be on filtering by the layer's addressing mechanism, mac addresses for ethernet, IP address for l3, and port for l4. This test case will not focus on edge cases and will instead attempt to test for a basic level of functionality. A more detailed description of what packets will be created can be found in the Proposed Pattern Generation section.

Actions

This test case or series of test cases will focus on processing a small group of packets, sent many times, and performing actions on them. The most heavily tested will be the ones that have side effects outside of testpmd, such as changing a field or sending out a specific port. Actions for which there is neither logging in testpmd nor observable side effects in packet structure will not be tested.

Edge cases

These test cases will be designed to test either the capabilities of the parser, the underlying driver, or hardware. These will include cases such as adding an excessive amount of VOID layers to a pattern, using RAW to jump to before the start or after the end of a packet, nesting tunneling protocols, attempting to match more bytes than the maximum MTU of the network card, and odd combinations of protocols that would normally seldom be used. These test cases will attempt to test things that make no sense in context but are valid flow rules. These test cases are not expected to pass on all NICs, given that they will probably run into hardware constraints.

Negative Test Cases

This test case will contain rules that are expected to be rejected as invalid, such as;

- leaving off the " / end " from a rule
- matching UDP as the outermost layer of a packet
- omitting the "/" character from rules
- matching invalid addresses (eg, IPv4 source address is matched to an IPv6 Address)
- A string of random characters (pre-generated to avoid a valid rule being created)

Test Procedure

1. Remove all current rules
2. Add the rule to testpmd
3. Pass if the rule is rejected via a value error code (i.e. rule not supported by the hardware/driver)
    a. These tests are meant to determine whether claimed functionality works as expected, not determine unclaimed functionality
4. Send a packet matching the rule
5. Fail if the side effects of the action were not detected
6. Send a packet which doesn't match the rule
7. Fail if the side effects of the action were detected

Proposed Pattern Generation

The following assumptions are made to avoid having to profile all of the drivers compatible with dpdk. If you find one of these assumptions to be invalid, please let me know to the best of your ability, as I am aware that some of this information may be proprietary.

Assumptions Made

1. That the bytes below the lowest match will not cause a problem with matching
    a. Ex: In a packet like Ethernet / IPv4 / TCP / HTTP, with a rule matching a given IPv4 address, then TCP and HTTP may be ignored,
    b. Ex2: After successfully testing matching IPv4, it is assumed defining a pattern on top of IPv4 (i.e. TCP Port) can be tested independently
2. Scenarios such as triple-stacked VLAN tags or nesting tunneling protocols are rare enough to not need to be tested in-depth.
    a. By nesting tunneling protocols, I mean situations like having 2 VXLAN headers in a single frame, forming a packet like
    Ether / IP / UDP / VXLAN / Ether / IP / UDP / VXLAN / Ether / ...
3. There is a separation between packet layers
    a. By this, I mean that the code to parse lower layers is shared between protocols in the same layer if the lower layer protocol can be used under both protocols. For instance, the parsing code for TCP is the same whether it is under IPv4 or IPv6. This does not need to be strictly true, but the methods should be equivalent.
    b. This is done to avoid needing to test every single possible packet, as it allows the assumption that a given protocol only needs to be tested with one of its possible parents.

RTE Flow Tokens Used

Protocols

- UDP
- TCP
- SCTP
- IPV4
- IPV6
- ETH
- VLAN
- VXLAN
- GRE
- ICMP
- ICMP6

Pattern Operators

- MARK
- META
- TAG
- ~~FUZZY~~
- INVERT
- ANY

Actions

- PASSTHRU
- FLAG
- DROP
- COUNT
- MAC_SWAP
- DEC_TTL
- JUMP
- MARK
- QUEUE
- RSS
- PHY_PORT
- PORT_ID
- OF_SET_MPLS_TTL
- OF_DEC_MPLS_TTL
- OF_SET_NW_TTL
- OF_DEC_NW_TTL
- OF_COPY_TTL_OUT
- OF_COPY_TTL_IN
- OF_POP_VLAN
- OF_PUSH_VLAN
- OF_SET_VLAN_VID
- OF_SET_VLAN_PCP
- OF_POP_MPLS
- OF_PUSH_MPLS
- VXLAN_ENCAP
- VXLAN_DECAP
- NVGRE_ENCAP
- NVGRE_DECAP
- RAW_ENCAP
- RAW_DECAP
- SET_IPV4_SRC
- SET_IPV4_DST
- SET_IPV6_SRC
- SET_IPV6_DST
- SET_TP_SRC
- SET_TP_DST
- SET_TTL
- SET_MAC_SRC
- SET_MAC_DST
- INC_TCP_SEQ
- DEC_TCP_SEQ
- INC_TCP_ACK
- DEC_TCP_ACK
- SET_TAG
- SET_META
- AGE

Generation Methodology

Pattern Protocol Stack

The base pattern tree will be generated by using the type system that was mentioned in Why an Exhaustive Test Suite is not Practical. This tree will be constructed from the bottom up of different pattern items at each stage, with all pattern items being associated with data about the next valid pattern items. These patterns will not have any parameters in them at this stage. Instead of working down the tree of protocols to generate all possible flow patterns, assumption 3 will be used to allow working up the tree of possible flow patterns. This means that each pattern item will have a list of valid parents and one will be arbitrarily chosen to be used as the parent when testing the pattern item. Assumption 1 will be leveraged to allow the placement of junk data in the payload of any data-carrying packets. This will result in a tree structure which can be iterated through to provide all of the required patterns.

After the base pattern tree has been generated, then the parameters for pattern items will be generated. These parameters will be pulled from a dictionary which contains mappings of parameter names to a predetermined subset of all possible values. This will serve to allow the generation of parameter values from a representative set without creating an excessive number of new rules. If this proves to result in too many rules, then sets of parameters may be hardcoded or reductions made in higher levels, such as not specifying the IP and MAC addresses if the current protocol is a level 4 protocol. If this proves too difficult or too resource-intensive to do on a per pattern basis, then it will be applied globally, with protocols having different behavior based on their height in the tree. This would allow the generation of a list of patterns that can then be processed in the test.

The end goal is to limit the total number of patterns to a reasonable amount, to keep the test time short enough to not bottleneck the pipelines.

Pattern Operators

Pattern operators are any pattern item that is not a networking protocol, such as VOID, RAW, INVERT, PF, or VF. Flow rules for these will be constructed by hand due to their wide range of side effects. These operators will then be configured with a representative subset of their parameters programmatically. Ideas for rules which would make a useful test case either due to interaction between operators or possible edge cases resulting from odd parameter values would be welcome.

Actions

Actions will be handled in much the same way as Pattern Operators, with a set of simple packets being used to verify expected behavior being designed by hand.