Toward Dynamic Configuration of Stateful Dataplane

Xin Wang†, Qiao Xiang‡, Y. Richard Yang‡,
†Tongji University, ‡Yale University,
13xinwang@tongji.edu.cn, {qiao.xiang, yry}@cs.yale.edu,

I. MOTIVATION

In this section, we will motivate the direct communication between middleboxes and switches (XXX Name?) with a firewall example as shown in Figure 1.a (XXX). The topology of the firewall example consists of one middebox (*i.e.*, firewall, fw), two hosts (h_1, h_2), and several switches (a - e). The firewall fw will identify whether a flow is $GOOD$ or BAD based on the 5-tuple of the flow (*i.e.*, srcIP, dstIP, srcPort, dstPort, protocol). And the network operator wants to enforce the following policy: For any flow from host h_1 to host h_2, if the flow is identified as $GOOD$, then it should pass through the switch c, otherwise pass through the switch d. Such policy can be specified by a high-level language as the following (the firewall program):

```
L1: mFW = middlebox("firewall")
L2: path1 = H -> c -> H //waypoint c
L3: path2 = H -> d -> H //waypoint d
L4:
L5: def onPacket(pkt):
L6:     if pkt.srcAddr = h1 & pkt.dsrAddr = h2:
L7:         if mFW.handle(pkt) = GOOD:
L8:             return path1
L9:         else:
L10:            return path2
L11:    else: return DROP
```

The firewall program follows the SNAP (XXX) syntax with the following extensions: 1. Add packet handling functions for available middleboxes in the network. For example, line 1 defines a middlebox variable mFW with "firewall" as a label to map to the middlebox fw in the network; line 7 invokes the packet handling function of mFW which can return the result of the identification of the flow (*i.e.*, the 5-tuple of the packet); 2. Support route algebra as the return. The route algebra (XXX) is used to give the constraints of a path in the network. For example, line 2 and 3 defines two paths with waypoints which means the path must pass through the waypoint.

Given the firewall program and the topology, the problem is how to setup the configuration of datapaths (*i.e.*, the flow rules) to enforce packets are handled as the program specifies. The result of the existing work (*i.e.*, SNAP) to achieve the setup of configurations is shown in Figure 1.b (XXX). Specifically, as shown with the two generated paths $path_1$ and $path_2$, for all the packets from h_1 to h_2, they should first pass through the firewall and then the required waypoints. The result gives the correct paths for packets from h_1 to h_2 following the firewall program. Note that the original SNAP cannot support the route algebra and it considers the network with the OBS (One-Big-Switch) model. Therefore, ~~to deploy the program,~~ one implementation is to add two virtual middleboxes at c and d. Then, for the program side, it converts to the following where the variable mC (mD) specifies the virtual middleboxes at c (d). (The locating of sources and destinations of generated paths relies on hints from network operators who specifies the mapping from ip addresses to ports ~~in OBS~~ (XXX).)

```
...
L7: if mFW.handle(pkt) = GOOD:
L8:     mC.handle(pkt)
L9: else:
L10:    mD.handle(pkt)
```

Though SNAP can give the correct configuration, it does not fully utilize the network resource/capabilities. Specifically, considering the stateless property of the firewall middlebox (*i.e.*, the identification is only based on the 5-tuple of packets), after the identification of a flow, the result can be "cached" in switches rather than still making packets passing through it.

Figure 1.c (XXX) gives an alternative design which leverages the direct communication between the firewall and (stateful) switches. We assume the stateful switches logically have two tables: one is a state table (*i.e.*, $match \rightarrow state$) and the other is a match table (*i.e.*, $match + state \rightarrow action$). A packet first enters the state table to get its corresponding state (by matching its packet fields) and then enters the match table to get the action. By allowing direct communication between the firewall and switches, the firewall can install/modify rules in the state table of switches. For example, it can install a rule: $srcAddr = h_1 \& dstAddr = h_2 \rightarrow NO-NEED-FW$ to the state table which indicates a packet from h_1 to h_2 has a state $NO-NEED-FW$.

The design has 5 steps: 1. The first packet enters a and then is forwarded to fw (by getting a state $NEED-FW$); 2. fw identifies the packet as $GOOD$ and then modifies its state at b to $GOOD$; 3. fw sends the packet to b where the packet gets state $GOOD$; 4. fw modifies the packet's state at a to $NO-NEED-FW$; 5. The following packets which have the same match fields with the 1st packet enter a and then are all forwarded to b. Compared with the SNAP, the alternative design ~~which leverages the direct communication between the firewall and switches,~~ may obtain a higher throughput as flows

DDC
based on
as
can
a-e
location
in the network
H: any host
a
capability
and has
dynamic configuration for (stateful) switches
indicate
dynamic configuration
(stateful) switches
NNF
line 2 (3) specifies the path
must pass through c (d) as its
does not need to forward to FW.
arrives at
waypoints constraint.
NF
The configuration result of the
existing work (i.e., SNAP) is shown
NNF
result of SNAP
path1 (path2) is used to forward packets
that are identified \emph{GOOD}
(\emph{BAD}) by the firewall.

do not need to pass through the firewall except the first packets. However, the design not only needs to give the paths but also the behavior of the firewall, *i.e.*, modifying what state at which switches (like step 2 and 4). Then, the problem ~~need to solve is~~: Given a program and a topology, how to automatically setup the configurations of switches and middleboxes to improve the performance by leveraging the direct communication between middleboxes and switches?

II. SYSTEM

In this section, we will give the system overview including the translation from programs into flow rules at switches and configurations of middleboxes (*i.e.*, how to interact with switches). We will start with the extension of programming model from SNAP (XXX) and then describe each component of the system.

A. Programming Model

As described in the motivation section (XXX), we consider the SNAP syntax as the base of our program's syntax. Since we do not focus on the state placement problem (which is the major component of SNAP), our programming model only consider two kinds of statements that can apply to packets: test of packet match fields (*e.g.*, line 6 in the firewall program) and packet handling of middlebox (*e.g.*, line 7 in the firewall program). The former one is the same with the SNAP ~~syntax~~ and the latter one returns the result of packet handling of a middlebox. Another difference with SNAP is that we support route algebra in programs (*e.g.*, line 2 and 3 in the firewall program). ~~For the generality, we consider the same syntax with Trident (XXX).~~

Also, as only stateless middleboxes can leverage stateful switches as cache of its packet processing, the system needs to identify whether a middlebox is stateless or not. For simplicity, we make the programmer give the hint. In the program, when initializing a middlebox variable, the program should specify whether it is stateless or not. The following gives an example that how to give a hint at the middlebox variable initializing statement.

```
mFW = middlebox("firewall", "stateless")
...
if mFW.handle(pkt) = GOOD:
```

B. Middlebox Behavior

We consider a middlebox as a packet handling function that can return a result for the incoming packet. For the stateless middlebox, if two packets have the same 5-tuple match fields, then they always have the same results. For the stateful middlebox, this cannot be guaranteed as its packet processing depends on the state in the middlebox.

For the interaction between a middlebox and a switch, we consider the following model. A switch can send a packet to a middlebox for the processing, and after the processing the middlebox can send the packet back and update a flow's state (based on the result) at any switch. (The flow has the same 5-tuple with the packet.) This process is very similar with the interaction between OpenFlow switches and the controller which includes packet-in messages, packet-out messages, and OpenFlow rules updates. Note that by using tunnels, any switch can send a packet to any middlebox, and also the inverse. Though the interaction is simple, there are several details need to consider. We will talk about the details in the next section.

C. Workflow

The workflow is shown in Figure 2 (XXX). Step 1: Compute the Decision Graph (DG) from the program. ~~As an internal representation of the program, the internal nodes in DG act are the tests of packets and the leave nodes are the paths (*i.e.*, path constraints) for packets.~~ Step 2: Based on the DG and the topology, compute optimal paths (*i.e.*, concrete paths instead of constraints) for packets. Step 3: Generate datapath configuration (including switch-middlebox tunnels) ~~for all the nodes (switches and middleboxes)~~ in the network.

III. IMPLEMENTATION

In this section, we will first provide the details of DG, and then discuss the path computation based on the DG and topology. Finally, we will discuss datapath generaiton from the paths.

A. Decision Graph

The decision graph is used to capture the trace of every packet applied by the program. The trace is the decision dependencies (*i.e.*, packet fields test statements and middlebox processing statements and their results) of a packet and its returned path (*i.e.*, path constraints). In this paper, for simplicity, we only consider the waypoints constraint (*i.e.*, a sequence of nodes in the network that packets should pass through). We define a DG as a directed acyclic graph (with a single root) that has three types of nodes: packet test nodes, middlebox operation nodes, and action nodes. The first two nodes are the internal nodes of DG while the last one is the leave nodes. An out-edge of a node can specify a range of packets (if the node is a packet test node) or a result from a middlebox (if the node is a middlebox operation node). For example, the DG of the firewall program is shown in Figure 3 (XXX). Note that we do not focus on the computation of DG from a program which can be achieved by existing (compiler) work (XXX).

Given a DG, we denote the sequence of nodes and edges from the root to an action node (*i.e.*, path p) as the trace of p ($T(p)$). Then, given a trace $T(p)$, we denote the middlebox operation nodes of $T(p)$ as $M(p)$. As middlebox operation nodes represent the packet handling of middleboxes, to not form loops, every middlebox operation node can only appear once in $M(p)$. Also, by extracting middlebox operation nodes that represent stateful middleboxes from $M(p)$, we have a subsequence of $M(p)$ that every node is for stateful middlebox operations (denoted by $M^s(p)$).

B. Path Computation

As the leave nodes in DG are path constraints, path computation targets to compute the concrete paths for these leave nodes with a performance objective ~~(in this work, we consider the total throughput)~~. Before path computation, we first give a packet forwarding model with DG in the network as the following. As shown in Figure 4 (XXX), given a path ($a \rightarrow b \rightarrow c$), logically we consider every switch in the path has a DG where P indicates a packet test node, M indicates a middlebox operation node, and A indicate an action node. The A node with red color represents the path. And a flow ~~targeting~~ for the path should be (logically) processed by middlebox M_1 and M_2 sequentially. The first packet of the flow (arriving at a) starts to traverse the DG as the red line in the DG. When the packet meets a middlebox operation node, it will be sent to the corresponding middlebox through a tunnel. After the processing, the middlebox will send back the packet and also set the state of the flow for its corresponding middlebox operation node at every switch along the path. (If the middlebox is stateful, then it does not set any state but make the packet carry the state.) After the traversal of the first packet (*i.e.*, arriving at the leave node along the red line), the packet will get the path and be forwarded along the path. As states have been set at corresponding middlebox operation nodes (or carried by the packet for stateful middleboxes), the packet does not need to be sent to middleboxes again at the next switch. We denote the forwarding of the packets do not involve the stateless middleboxes as the stable forwarding.

As the initial process, for action node in the DG, system should identify its source and destination nodes in the network. (This can be achieved by the hints from network operators as (XXX).) Then, add the source and destination nodes to the corresponding waypoints constraint. Also add its $M^s(p)$ to the waypoints constraint as the processing of stateful middleboxes should have a correct order. We consider the constraint as a set of node pairs (a, b) which means packets must pass through a and then b. (Note that a and b can be source/destination nodes.)

Based on the model, as the forwarding of the first few packets has little influence on the performance of its flow, a simple path computation (that only considers the stable forwarding) would be the following. We consider the objective of the network is to maximize the total throughput ~~(*i.e.*, maximize $\sum_i b_i$). And the constraints are only for the path constraints (for a starting point, we only focus on the ordered waypoints constraints) specified by the programmer (*i.e.*, network operator).~~ The formal definition of variables can be found at Table ??.

And the constraints can be found at Table ??.

However, this simple solution can get optimal paths only if all the middleboxes are stateless. If there is a stateful middlebox, the constraints of stable forwarding should not only consider the ~~ordered~~ waypoints constraints specified by the programmer as it also must pass through the stateful middlebox.

Variable	Description
u_i, v_i	The source and destination nodes of path i
~~N~~	All nodes in the network
E	All edges (an edge e: $(e.src, e.dst)$) in the network
m_e	The maximum bandwidth of e
W_i	A set of node pairs of path i
x_e^i	The edge e is selected by path i
b_i	The bandwidth of flows using path i

Constraint	Description
$\forall i, Path(u_i, v_i, E)$	A path i exists in E
$\forall i, E_i = e * x_e^i \forall e \in E$	Compute the all the edges in path i
$\forall (a_i, b_i) \in W_i, Path(a_i, b_i, E_i)$	Ordered waypoints constraint for path
$\forall e \in E, \sum_i b_i \leq m_e$	Bandwidth constraints of all edges

Then, people may add another constraint as the following: $\forall (a_i, b_i) \in M^s(p_i), Path(a_i, b_i, E_i)$. This constraint enforces that the optimal paths must pass through all the stateful middleboxes in a correct order (*i.e.*, $M^s(p_i)$). This seems correct as now the stable forwarding includes the stateful middleboxes.

However, this is wrong. Considering the example as shown in Figure 5 (XXX). As the middlebox M_3 is stateful, the new added constraint enforces that the computed optimal paths should consider M_3. Then, the result could be that M_3 is behind the branch of waypoints b and c. In this case, when a packet arrives at a, it cannot decide how to forward to M_3 as which path to select depends on the result of M_3.

To resolve the issue in Figure 5 (XXX), one solution is to add constraints (M_3, b) and (M_3, c) to the A_1 and A_2 respectively. Then the computed paths in Figure 5 (XXX) cannot exist as they does not satisfy the constraints. The idea is to enforce a packet must get results of stateful middleboxes and then select the correct path.

Though the idea is simple, a straightforward implementation may cause excessive constraints for paths. For example, a stateful middlebox operation node (M) has two path constraints A_1 (must pass through a, b) and A_2 (must pass through a, c) as its child node in a DG. Based on the idea, we can add (M, a) to both A_1 and A_2 waypoints constraints, and the computed paths are guaranteed to meet the constraints. However, this is only a sufficient condition as some other conforming paths are ~~also~~ excluded (*e.g.*, a to M to b). ->

Now we give an algorithm to enforce all the paths are correct and no conforming paths can be excluded. The high-level structure of the algorithm is to do a depth-first traversal for a DG. The traversal only considers the stateful middlebox operation nodes and action nodes (and traverses a stateful middle operation node only after all its child nodes are finished ~~for traversal~~). When traversing a stateful middlebox operation node, we can get all possible waypoints constraints under its decision. Therefore, the processing of the node is to update these constraints (each of which can be modeled as a directed acyclic graph, DAG). As discussed before, we do not want to add excessive constraints. And the solution is simple: When processing a middlebox node M, if all the no-incoming-edge

nodes of M's possible constraints (*i.e.*, DAGs) are the same, then skip these nodes recursively until not and then add edges (M, x) to each of DAGs where x is the node of the location of the skipping process stopped. The insight is that, skipping a node is correct if and only if the node is the next waypoint node for *all* the possible paths. For the example in Figure 6 (XXX), the node M has four possible path constraints (DAGs) and a is skipped when updating them.

C. Datapath Generation

After the path computation, there are concrete paths for all the leave nodes in DG. We ~~first~~ follow the packet forwarding model described previously. Then, all the switches in the network have the same DG. It is easy to observe that if a switch does not belong to a path, then the corresponding leave node of the path can be removed (and then the no-out-edge internal nodes). Also, we can convert the path in a leave node to the next hop in the network.

After this reduction of DG for all the switches, we can still remove nodes. Now return to the firewall example. As switch a belongs to both paths $path_1$ and $path_2$, the two leave nodes cannot be removed. However, after we convert the paths to next hops, we find that the next hops of two leave nodes are the same, *i.e.*, switch b in the network. This means the middlebox operation node has no meaning for the switch a since whatever the result of the node is, the next hop does not change. Therefore, we can remove the node and replace with the leave node with the next hop b. Figure 7 (XXX) illustrates the process. For the switch side, based on the DG, it can generate tables and flow rules easily with the multi-table pipeline structure. As for the middlebox side, if it is stateless middlebox, the only datapath it needs to care about is switch-middlebox tunnels which also can be easily generated. For the stateful middleboxes, as they belong to computed paths, the forwarding capability is need for them, which means it can be viewed as a switch node with a stateful function (*i.e.*, the corresponding stateful packet handling functions can apply to packets). ~~Note that~~ when a packet meets a stateful middlebox node in DG of a switch, and the node cannot be replaced with a leave node, then any next hop under the node is acceptable since it must eventually arrive at the middlebox and any path from the switch to the middlebox must follow the path constraints.

The next issue is how a stateless middlebox updates a state in other switches. It needs to guarantee that for any targeting switch, the update message should arrive earlier than the packet. For example, in the firewall example, only the step 2 is finished, step 3 can be executed. A simple solution is making the packet carrying the update message. Different with the stateful middleboxes, this update can be set to the corresponding node (table).

IV. Evaluation

In this section, we will first demonstrate the benefits of DDC from two aspects: latency and total throughput, and then evaluate its performance by showing the execution time of path computation part. All evaluations are run on an 3.5 GHz Intel i7 processor with 16 GB of RAM running Mac OSX 10.13.

Methodology: ~~We consider two aspects (*i.e.*, latency and total throughput) to show the benefits of DDC.~~ First we generate a random topology with 25 nodes and 50 edges. For every edge, we set two random values to it as its latency (5 - 10ms) and bandwidth (50 - 100Gbps) ~~respectively~~. To model a flow in the topology, we randomly choose two nodes from the topology as the source and destination nodes of the flow. And a flow can have a sequence of nodes (other than its source and destination nodes) in the topology as its required ordered middleboxes for packet processing. (We add a constraint to select these middlebox nodes that the number of neighbors of a middlebox node must equal to two as typically a middlebox does not have route selection capability, *i.e.*for any packet, it only has one output interface.) As a comparison of DDC, the traditional approach does not distinguish whether a middlebox is stateful or stateless. Therefore, when computing a path for a flow in the tradition way (*i.e.*, do not apply DDC), it requires the path must pass through all the middleboxes in a correct order ~~as specified in the flow's requirement~~. And when computing a path for a flow in the DDC approach, we random choose a subset of flow's required middlebox nodes as stateful middleboxes since for DDC approach, stateful and stateless middleboxes are handled in different way.

To show the benefits from the latency aspect, we only consider one flow and differentiate its number of required middleboxes ~~as different experiments~~. And the target is to find an optimal path to minimize the latency for the flow. Then, we compare the results (*i.e.*, the minimal latency) between applying DDC and not.

To show the benefits from the total throughput aspect, we consider multiple flows and all flows have the same required middleboxes. We also differentiate the number of middleboxes ~~as different experiments~~. And the target is to find optimal paths that have maximum total throughput. Then, we compare the results (*i.e.*, the maximum total throughput) between applying DDC and not.

To evaluate the performance of DDC, we compare the execution time of the path computation part for both approaches (*i.e.*, applying DDC or not).

Result: The result is shown in XXX.