Inter-Flow Consistency: Novel SDN Update Abstraction for Supporting Inter-Flow Constraints

Weijie Liu*, Rakesh B. Bobba†, Sibin Mohan‡ and Roy H. Campbell*
*Dept. of Computer Science, University of Illinois at Urbana-Champaign, IL USA
†School of Electrical Engineering and Computer Science, Oregon State University, OR USA
‡Information Trust Institute, University of Illinois at Urbana-Champaign, IL USA
{wliu43, sibin, rhc}@illinois.edu, rakesh.bobba@oregonsstate.edu

***Abstract*—Software Defined Networks (SDN) gained a lot of attention both in industry and academia by providing flexibility in network management. With decoupled control and data planes, operators find it more convenient to configure and update their networks. However, transitional states of SDN during updates may be a combination of the old and new network configurations. This may lead to incorrectness in forwarding behaviors and security vulnerabilities – this is particularly important in certain safety-critical applications.**

In this paper, we argue for a novel abstraction for network update consistency, *inter-flow consistency*, that accounts for relationships and constraints among different flows during network updates. For two basic inter-flow consistency relationships, *spatial isolation* and *version isolation*, we propose an update scheduling algorithm based on dependency graphs, a data structure revealing dependency among different update operations and network elements. We also implement a prototype system with a Mininet OpenFlow network for spatial isolation and undertake a preliminary performance evaluation of our solution.

I. Introduction

Network infrastructure and flows evolve constantly. Network operators need to reconfigure their networks frequently to support dynamic access control, traffic engineering, security updates, and infrastructure maintenance and updates among other things. These updates need to be carefully planned and scheduled to minimize disruptions. However, it is hard to guarantee that such updates will not result in problems. Even though there exist verification tools to check the state of the network [2], [10], [16], transitional states caused during network updates are still problematic.

Software-Defined Networking (SDN) [18] changed the way we view networking, especially network management and maintenance, mainly by decoupling the control plane from the data plane. SDN provides lot of flexibility and powerful capabilities to many network applications, *e.g.,* virtual machine migrations [3], traffic engineering [21], access control [20], server load balancing [13]. SDN also increases the convenience for network operators – both for setup, management and later maintenance and updates. But even with SDN in place, there is no guarantee regarding the *correctness* of traffic flows and network state during the update process [15], [22] even when the initial configuration and final configuration are verifiably correct. This is because even though SDN provides a centralized place, namely, the network controller to manage and update the network, the controller itself has to distribute the configuration to switches and routers that are distributed. As such without additional mechanisms configuration changes are not synchronous across the network infrastructure. Lack of consistency during network update process can not only adversely impact the stability and availability of the network (by causing transient black holes and loops) but also its security. For example, incorrect routing of packets during network transition may cause packets to go around a security middlebox such as a firewall or a network intrusion detection system and create a security hole.

Early work addressing this problem [22] proposed two correctness abstractions for network updates in SDNs: *per-packet* and *per-flow* consistency. Per-packet consistency means that each packet in the network will be processed either by the old configuration or the new one but never a mixture of the two. Per-flow consistency generalizes per-packet consistency and guarantees that each flow in the network will be processed by the old configuration or the new one, but not the mixture of the two. This work also proposed mechanisms to implement these update abstractions based on the OpenFlow [18] protocol. Since then a lot of research effort has gone into network systems that can guarantee update correctness for different application scenarios building on these two update abstractions.

In this work we argue that per-packet and per-flow consistency alone are not sufficient for meeting requirements imposed by security and reliability. Most networks have multiple flows and there are often cases where we would like to preserve some relationships between the flows during network updates to maintain security and reliability. Per-packet and per-flow consistency abstractions do not account for the relationship between flows and are thus not sufficient to meet such security and reliability requirements. For instance, in control networks of power systems, it is desirable to separate certain critical real-time control flows from engineering flows, *i.e.,* ensure that they never share a link, for reliability of real-time operations. Similarly, it is desirable to isolate two flows from each other if one is a back-up flow for the other. As another example, in data centers it may be necessary to separate flows belonging

SENT '15, 8 February 2015, San Diego, CA, USA

http://dx.doi.org/10.14722/sent.2015.23006

to tenants from competing organizations to provide security isolation required by SLAs.

Another common scenario is where a (distributed) application imposes a relationship among network flows that needs to be preserved during the update process. Stateful firewalls are one example of application where their correct operation can depend on relationship between flows. In such cases, processing flow updates using per-packet or per-flow consistency may lead to a state of the network that is a combination of old configuration for some flows and new configuration for others leading to unexpected results, *i.e.,* forwarding loops, packet loss and incorrect application execution [5].

In this paper, we argue for a novel update abstraction, namely, *inter-flow consistency* that accounts for relationships and constraints among different flows is needed to address this problem. We present two special cases of inter-flow consistency, namely *spatial consistency* and *version consistency*, and design algorithms based on dependency graphs of [7] to achieve inter-flow consistency during SDN updates. While we primarily motivate the need for such an abstraction through security and reliability requirements the abstraction and the proposed mechanisms are generally applicable.

II. Inter-Flow Consistency

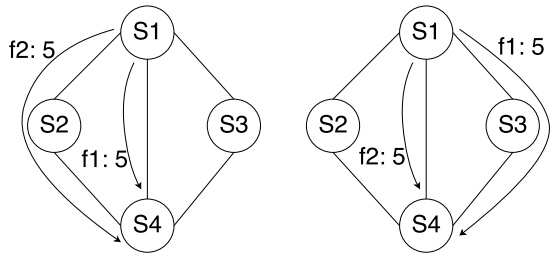
Inter-flow consistency for updates is a guarantee that specified inter-flow are preserved during network updates[1]. While there can be many kinds of inter-flow constraints that are needed, we discuss two specific types in this work that are motivated by security and reliability requirements: *(a)* spatial isolation and *(b)* version isolation.

A. Spatial Isolation

Spatial isolation represents the requirement that certain flows are not allowed to share a link or a switch before, during and after an update for security and/or reliability reasons. For instance, if a flow has certain criticality requirements (it carries control messages in a power grid substation) then sharing links on its path with another flow that carries, say debugging or engineering traffic, may result in problems for the former flow. For instance, if there is a surge in information being sent over the engineering flow because of say some firmware upgrades then we don't want critical control messages suffering delays and/or dropped packets. Other examples could include situations where hackers could try to use information about their own flows (*e.g.,* round-trip times or packet loss rates) to infer details about the critical flows. In our work, we assume that the original flow configurations were consistent with these requirements and the new, updates flows will also satisfy them. The problem arises during the *transitional states*. Hence, we need mechanisms to ensure that the spatial isolation requirements are satisfied at all points during the update phase.

Consider the simple example in Figure 1. Figure 1(a) shows a network with 4 switches and two flows, f_1 and f_2. Lets assume that flows f_1 and f_2 are not allowed to share a same link; in order words, f_1 and f_2 should demonstrate spatial isolation properties. In this example, each the flow

[1]It is assumed that these constraints are satisfied by both the initial configuration and the target configuration



(a) original state (b) target state

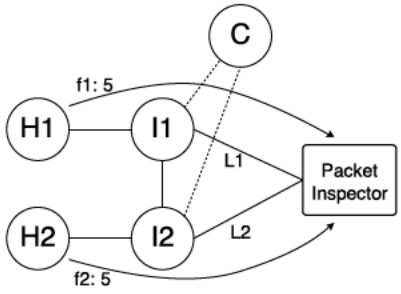
Figure 1. An example for spatial relationship of flows

consumes 5 units of bandwidth while the bandwidth of each link is 10 units. Originally, f_1 passes through the link, S_1S_4, and f_2 passes through S_1S_2 and S_2S_4. Now if we are to update the network to a new state (shown in Figure 1(b)) so that f_2 passes through S_1S_4 and f_1 passes through links S_1S_3 and S_3S_4. It is clear that the old as well as the new configurations of this network can guarantee spatial isolation. However, due to the asynchronous nature of flow updates we might have a transitional state where f_2 is updated before f_1. In that case, both the two flows pass through the same link, S_1S_4, for some finite time violating the inter-flow consistency requirements. Thus, update mechanism must guarantee that the spatial relationships (if any) between any two flows is preserved during the update process.

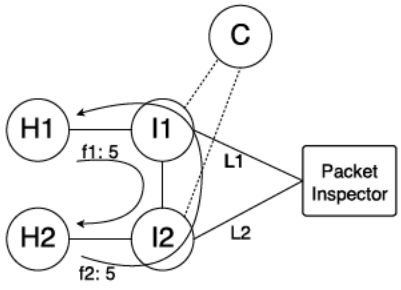
B. Version Isolation

Version isolation means that packets from different related flows cannot be processed by two different *versions of flow rules* during its passage through the network. This can happen because the network updates for certain flows have not been completed before they start routing packets. Imagine a scenario with 2 flows, A and B; let the states of Flow A before and after an update be R_{A1} and R_{A2}, respectively. Let the states of Flow B before and after an update as R_{B1} and R_{B2}, respectively. The network can have $R_{A1}R_{B1}$ or $R_{A2}R_{B2}$, but not $R_{A1}R_{B2}$ or $R_{A2}R_{B1}$ at any point in time. We refer to this as *version isolation*.

An example is shown in Figure 2, which is a revised version of a case in [5]. H_1 and H_2 represent two hosts each of which sends out a flow (f_1 and f_2, respectively). Each flow consumes 5 units of bandwidth while the bandwidth of each link is 10 units. There are two ingress switches, I_1 and I_2, with a controller, C. Both of the switches are connected to a server running a packet-inspection application. At first, both of the two hosts send some verification packets to the inspector (shown in Figure 2(a)). After inspection, the application can ask the controller to modify the rules in the 2 switches so that the two hosts can communicate with each other through $I1$ and $I2$ (shown in Figure 2(b)). However, the forwarding rules in the two switches may be not updated at the same time. Imagine that if the rules of f_2 have been updated and f_2 is forwarded to H_1; but the updates of the rules for f_1 have not been finished yet. Receiving packets from H_2, H_1 might think that the packet inspection is completed and then transmits normal application packets other than verification packets to H_2. However, since the rules of f_1 have not been updated yet, these packets will be



(a) original state



(b) target state

Figure 2. An example for temporal relationship of flows

forwarded to the inspector–an unexpected outcome. Hence, we need to guarantee that new configurations and old ones should not exist at the same time for the two flows. Note that while this can be rare, an extreme case of version isolation is one where all the flows in the network require version isolation. That is, all the packets in the network should be processed by either the initial flow rule configuration or the target flow rule configuration but not a mix. If it is assumed that the initial and final flow configurations are designed to satisfy necessary inter-flow constraints then in this extreme case version isolation implies inter-flow consistency.

C. Assumptions

In this paper, we assume that there always exists a correct order of update operations. This correct order can preserve bandwidth guarantees, spatial isolation and version isolation during network updates. Admittedly, this assumption may be violated in real world, *e.g.,* by adding elephant flows to a network that already faces significant congestion, deadlocks among different flows – each of which holds a link and waits for others' links, *etc.* In the future, we may integrate existing solution [7] to relax these assumptions.

III. Approach

A. Dependency Graph

Our algorithm is based on an approach similar to dependency graphs [7]. A dependency graph represents update operations, resources and paths as well as the dependencies among them. In the original version of dependency graph [7], there are 3 types of nodes: *resource nodes*, *operation nodes* and *path nodes*. Resource nodes (shown in rectangles) represent the resources in the network, *e.g.,* link capacity and memory space of a switch. The number in a resource node represents the amount of available resources of that node. Operation nodes (shown in circles) represent addition, deletion, or modification of a forwarding rule. Path nodes (shown in triangles) represent a group of operations and resources related to a certain path.

A dependency graph has directed edges of different types. The edges from an operation node A to another operation node B means that B can not be scheduled before A completes. The edges between resource nodes and operation nodes represent a *resource dependency*. An edge from a resource node to an operation node represents that the operation needs this amount of available resource. An edge from an operation node to a resource node represents that this amount of resource will be freed by that operation. Similarly, an edge from a path node to a resource node represents a certain amount of resource will be freed by the deletion of that path. An edge from a resource node to a path node represents the addition of that path will consume a certain amount of resource. The edges between operation nodes and path nodes represent the proper schedule order of the two. An edge from a path node to an operation node represents that operation cannot be scheduled until P is removed. An edge from an operation node to a path node represents a path cannot be used until that operation completes.

B. Inter-flow Scheduling

1) Spatial Isolation: In order to represent the flow isolation relationships, we define a new type of node, *mutex nodes* (represented using diamonds) that capture the isolation requirements between different flows. The concept of a mutex is originated from the resource mutex in operating systems, *i.e.,* mutex is *available* only when no one occupies it. A mutex node can be considered as a special resource node. We only have edges between path nodes and mutex nodes. An edge from a mutex node to a path node means that that path cannot be used until the mutex is freed. An edge from a path node to a mutex node means that the mutex will be released after the update of that path.

For example, We can generate the dependency graph shown in Figure 3 for the schedule problem in Figure 1. First, we need to calculate the update operations shown in Table I by comparing the old network state and the new one. Path Node p_1 represents the path of f_1 before updates while p_2 represent that of f_2 before updates; p_3 and p_4 represent the path of f_1 and f_2 after updates, respectively. There is a common link between p_1 and $p4$; thus, there is a mutex node representing the common link, S_1S_4, between p_1 and p_4. The edge from Node p_1 to S_1S_4 means after updates f_1 will not pass through the link, S_1S_4. The edge from Node S_1S_4 to p_4 means that after B's updates f_2 will pass through S_1S_4. In Figure 3, with *topological sorting*, it is clear that a proper update schedule is first to update p_1 and then to update p_4. A valid order of the update operations is $a \rightarrow b \rightarrow c \rightarrow d$.

2) Version Isolation: We adopt a two-phase update approach [6] to deal with version isolation. Imagine a situation where we have several flows with version isolation requirement. First, we pick one flow among those with version isolation requirement, and forward the others in that set to the controller for storage. Then we update the flow rules

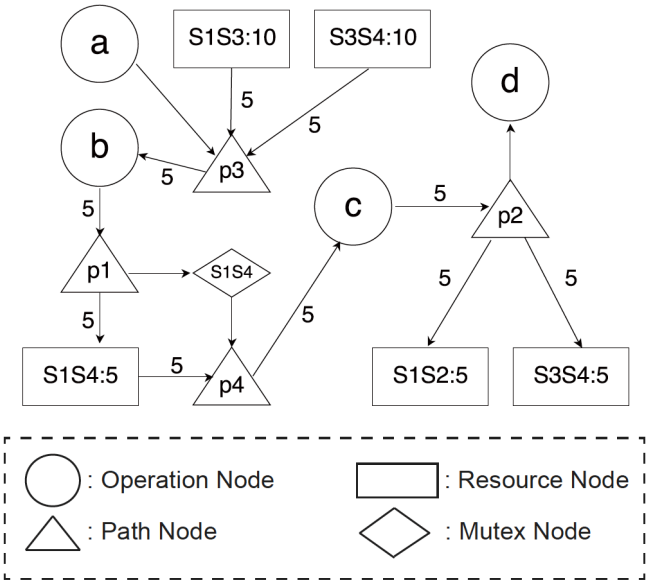


Figure 3. Dependency Graph from Figure 1

Table I. UPDATE OPERATIONS FOR FIGURE 1

ID	Entity	Update Operation
a	S_3	Add: forward f_1 to S_4
b	S_1	Modify: forward f_1 to S_3
c	S_1	Modify: forward f_2 to S_4
d	S_2	Delete rules of f_2

associated with this first flow; during this time, all the other flows that are version isolated with this flow are forwarded to the controller. Third, pick another flow among the version isolated set and update its rules in the network. Then the controller transmits the cached packets of the recently updated flow into the network. Repeat these steps until all the flows with version isolation requirement are updated. During this process, flows that did not have version isolation or other inter-flow consistency requirements could be updated in parallel. This approach is shown in Algorithm 1.

The intuition behind Algorithm 1 is that we update one flow at first and forward all the other flows in the version isolation set to the controller so that the flows with new rules will not be mixed with those with old rules in the network. Actually, at the time when some flows are being updated while others are cached in the controller, new versions coexist with "old version of rules" – they are just forwarding packets to the controller. However, since the packets with this old version of rules are away from the network data plane but at the control plane, we achieve version isolation to some extent. It is clear that Algorithm 1 costs overhead in the controller and introduces delays and availability concerns. Therefore, this approach is viable only in those cases where *(a)* the number of flows with version isolation requirements are reasonably small[2] and that *(b)* the amount of traffic forwarded to the controller is acceptable because the time of updates is relatively short. Another alternate method for version isolation is simply dropping all the relevant traffic during update [6] rather than to forward them to the controller. Again this can only work if the update process takes a very short time and in cases where flows can automatically recover.

[2]Small enough so that the update process for all flows can be completed without adverse impact on the flows.

Algorithm 1 Two-phase Update

Require: F: a set of flows for version isolation
1: $f_0 \leftarrow$ one of the flows
2: $F \leftarrow F - f_0$
3: Forward all the traffic of F to the controller
4: Update the rules for f_0
5: **for** each flow, f, in F **do**
6: Update the rules for f
7: Stop forwarding f to the controller
8: Controller injects cached traffic of f into the network
9: **end for**

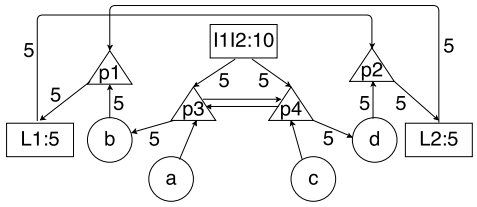
As an example, we can use the updates in Figure 2 to generate a dependency graph shown in Figure 4 with version isolation. First, we need to calculate the update operations shown in Table II by comparing the old network state and the new one. Then we can generate the dependency graph in Figure 4(a). The two arrows between p_3 and p_4 means that the corresponding two flows are required to be updated with version isolation. It is clear that we fail to find the topological order of the operations in 4(a) because of the existence of a loop. Based on Algorithm 1, we can first transmit the packets of f_2 to the controller and then update the rules of f_1. After that, we update the rules of f_2 and then the controller transmits the traffic of f_2 back to I_2. During this process, even though f_1 gets updated before f_2, there is no f_2 packet processed by the old rules while f_1 has been updated. The operations e and f represent the action of sending f_2 to controller and transmitting it back to the network, respectively.

A revised dependency graph is shown in Figure 4(b). A new operation, e, is added with many edges from e to the other operations related to the two paths. It means that the protocol should first forward f_2 to the controller before the rules are updated. This can prevent any packet loss due to updates. The other new operations, f and g, are added. The directions are from p_3 and p_4 to f, which means that only after new rules are installed can we inject the packets of f_2 back to our network. A valid order of the update operations is $e \rightarrow a \rightarrow b \rightarrow c \rightarrow d \rightarrow f \rightarrow g$.

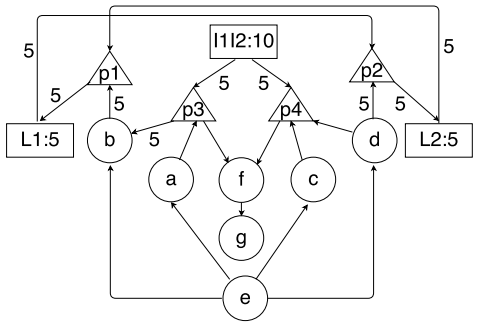
Table II. UPDATE OPERATIONS FOR FIGURE 2

ID	Entity	Update Operation
a	I_2	Add: forward f_1 to H_2
b	I_1	Modify: forward f_1 to I_2
c	I_1	Add: forward f_2 to H_1
d	I_2	Modify: forward f_2 to I_1
e	I_2	Modify: forward f_2 to C
f	I_2	Delete the rule of forwarding f_2 to C
g	C	forward all the cached traffic of f_2 to I_1

3) Scheduling Algorithm: We revise the previous scheduling algorithm [7] with considerations for inter-flow relationships. Because the dependency graphs represent the dependency relationships between update operations, the simple idea behind the scheduling algorithm is that the operations cannot be scheduled until they satisfy two conditions: 1) they have



(a) original dependency graph



(b) revised dependency graph

Figure 4. Dependency Graph from Figure 2

no ancestor nodes for operation nodes, and 2) the necessary resource are available. With the scheduling algorithm, we can divide the updates schedule into different rounds. In each rounds, we can schedule several update operations.

Another more fine-grained question is about the update order of the operations in each round. We adopt the same *critical-path* method as in [7]. In a classical DAG using nodes to represent several jobs in a project, a node's critical-path length (CPL) is the maximal value among the distances from this node to other nodes. In order to minimize the total completion time, we tend to schedule the job with the largest CPL first. In our dependency graph, we assign weights to the nodes: the weight w of an operation node is 1 while weights of resource, mutex and path nodes are 0. With this, we can calculate the CPL for each node i [7] recursively:

$$CPL_i = w_i + \max_{j \in children(i)} CPL_j \tag{1}$$

After sorting the nodes with their CPL in decreasing order, in each round of scheduling, we greedily schedule the operation nodes based this order. The scheduling algorithm is shown in Algorithm 2. At first, we generate and revise the dependency graph based on the input old and new network state (A network state can tell the path links and traffic load in each link of flows in a network). Then we leverage the dependency graph to divide the updates into several rounds; in each round we schedule the available operations in CPL decreasing order. Available operations are operations that have no ancestor operation nodes and can gain enough resources for updates. The correctness of Algorithm 2 is based on the assumption in Subsection II-C that there is a correct scheduling order of updates. In other words, there is no deadlock in the dependency graph. Thus, in each round, we can always schedule some operations and reduce the number of nodes in the graph. The algorithm will not result in infinite loops.

Algorithm 2 ScheduleUpdates(F, N_0, N_1, C)

Require: F: the flows whose forwarding rules to be updated
Require: N_0: the original network states of F
Require: N_1: the new network states of F
Require: C: update constraints(spatial or version isolation)
1: $G \leftarrow GenerateGraph(F, N_0, N_1)$
2: $G \leftarrow ReviseGraph(G, C)$
3: Calculate *CPL* for every node in G
4: Sort the operation nodes in the decreasing order of *CPL* and get a sorted order L
5: **while** $G \neq \emptyset$ **do**
6: **for** each operation node $O_i \in L$ **do**
7: **if** O_i has no ancestor operation nodes and can get the necessary resource for updates **then**
8: Schedule O_i
9: **end if**
10: **end for**
11: Delete scheduled operation nodes and corresponding path nodes as well as their edges
12: Delete resource nodes and mutex nodes without edges
13: Update the available amount in resource nodes
14: Wait for a time threshold for all scheduled operations to finish
15: **end while**

Algorithm 3 shows the algorithm of dependency graph generation. First, we create resource nodes in the graph. Second, by comparing the old and new paths of each flow, we can calculate the necessary update operations. Then edges are added between two nodes to represent the path-resource relationship or operation-resource relationship. Function $CreateEdges((s_1, d_1), (s_2, d_2), ..., (s_n, d_n))$ means creating an edge from each element in s_i to each element in d_i, respectively. A *floodgate* operation is the first update operation which will forward the packets from the old path to the new path.

Algorithm 3 GenerateGraph(F, N_0, N_1)

Require: F: the flows whose forwarding rules to be updated
Require: N_0: the original network states of F
Require: N_1: the new network states of F
1: $G \leftarrow \emptyset$
2: **for** each link, l, in N_0 and N_1 **do**
3: Create a resource node, v, representing l and its remaining capacity
4: **end for**
5: **for** each f in F **do**
6: $p_0 \leftarrow$ the old path of f in N_0
7: Create edges from p_0 to each related resource node
8: $p_1 \leftarrow$ the new path of f in N_1
9: Create edges from each related resource node to p_1
10: $o_f \leftarrow$ the *floodgate operation* of p_0 and p_1
11: $O_0 \leftarrow$ the operations for removing p_0
12: $O_1 \leftarrow$ the operations for creating p_1
13: $CreateEdges((o_f, p_0), (p_1, o_f), (p_0, O_0), (O_1, p_1))$
14: **end for**
15: **return** G

Since we would like to guarantee the inter-flow consistency during updates, we need to add the representation of spatial and version constraints into the dependency graph. For flows' spatial isolation, it is clear that we should create mutex nodes between two isolated paths which might occupy the same resource. On the other hand, it is much more tricky to guarantee flows' version isolation. We design a graph version of Algorithm 1 in Algorithm 4. First, we add an operation node, o_m, which forwards all the relevant flows to the controller. Edges are created from o_m to other operations of the relevant flows. After all the new forwarding rules are installed, we delete the rules of o_m and then inject the cached traffic into the network.

Algorithm 4 ReviseGraph(G, C)

Require: G: the original dependency graph
Require: C: update constraints(spatial or version isolation)
for each constraint c in C **do**
  if c is *spatial isolation* **then**
    for each common resource, r, shared by the flows specified by c **do**
      Create one mutex node, r_m
      Create edges between r_m and responding path nodes
    end for
  else if c is *version isolation* **then**
    $F \leftarrow$ the flows involved in c
    $f_0 \leftarrow$ one flow in F
    $F \leftarrow F - f_0$
    Create an operation node, o_m, representing forwarding F to the controller
    Create edges from m to all the parent operation nodes of the flows in F
    for each f in F **do**
      $p_1 \leftarrow$ the new path of f
      Reverse the direction of the edge between p_1 and the *floodgate operation* of f
      Create an operation node o_d, representing stopping forwarding f to the controller
      Create an operation node o_i, representing the controller injects the cached traffic of f into the network
      $CreateEdges((p_1, o_d), (f_0, o_d), (o_d, o_e))$
    end for
  end if
end for
return G

IV. Evaluation

A. Implementation Setup

We used Mininet [12] as our evaluation framework and created a traditional 3-level tree topology. There is only 1 core switch and 5 aggregation switches are linked to it. Each aggregation switch is linked to 5 ToR switches. In the experiments, we change the number of hosts linked to each ToR switches from 2 to 20. The bandwidth of each link is set as 1 Gbps. For each setup of hosts, we run our experiment 10 times.

For the control plane, we implemented a prototype system as an update scheduler between a shortest-path routing application and the **Ryu** controller [1]. We use Ryu's well-defined APIs to construct our control program in a little more than 1000 lines of Python code and run it on a PC with Intel i5-2400 3.1 GHz CPU and 16 G memory.

B. Flow and Constraint Generation

In our experiment, we define a sequence of packets with the same sender IP as one flow and each experiment has two phases. In the first phase, we generate a network state. A network state consists of randomly generated host-to-host pairs which have a flow between them. Each host in the network is only involved in one pair. In such a host-to-host pair, one host continuously sends out a sequence of UDP packets to the other. Then the shortest-path routing application will calculate and install the forwarding rules for each flow. At the same time, a module in our system records the path information for each flow. The flow rate is set as 1 Mbps.

Then we move on to the second phase where we generate a new network state. Then we carry out a brute force search: for a flow A and another flow B – if they are spatially isolated both in the old and new states, but not during the transitions (*i.e.,* A's path in the old state overlaps with B's path in the new state) then we assign a spatial isolation condition to A and B.

After we have the two network states and a list of spatial isolation situations, we run our update scheduler to carry out the necessary update operations (add, modify or remove flow rules in Mininet) while still maintaining the isolation requirements.

C. Experiment Results

Figure 5 shows the number of spatial isolation pairs (two flows are not allowed to pass the same link) and the total number of update operations as the number of hosts in the network increases. Standard deviations are also shown. Since we generated flows with host-to-host pairs, there is no surprise that the number of spatial isolation conditions and the total amount of update operations have a linear relationship with the number of hosts. Figure 6 shows the total running time T_t of our update algorithm, the time T_g for generating dependency graph and the time T_s for scheduling updates round by round. It is clear that $T_t = T_g + T_s$. All of the three time lengths increase linearly as the number of hosts increases. The total running time of our update algorithm is just 56 ms when there are 500 hosts, about 61 pairs of flows with spatial isolation and 1134 update operations. We also notice that on average 33% of the algorithm's running time is spent on the generation of dependency graphs while 67% is spent on the round by round scheduling.

V. Related Work and Background

SDN opens a new era of networking by decoupling the control plane and data plane [18]. With the advent of SDN, many works provide solutions for network verification. NetPlumber in [9] presents an efficient verification tool based on Header Space Analysis (HSA) for incremental compliance checking. Veriflow in [11] also provides fast network-wide invariants checking based on a concept of equivalence class. However,they focus on verification in the configured network

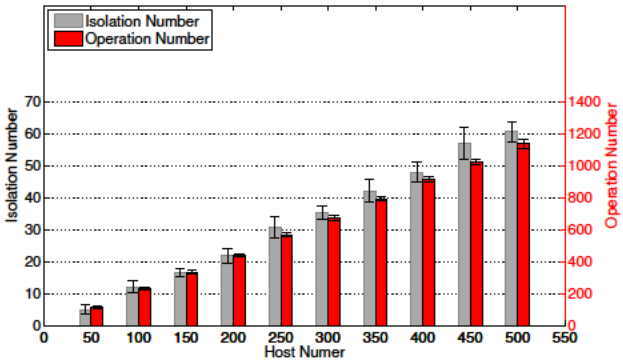


Figure 5. Number of Isolation Pair and Update Operation on different Host Number

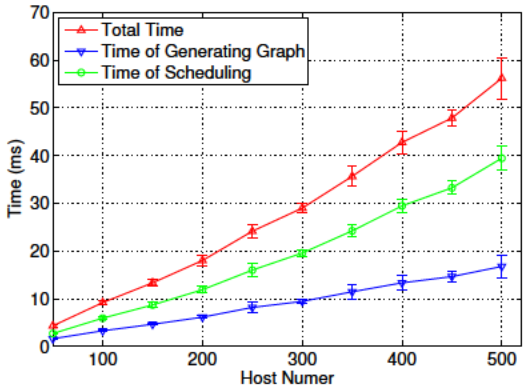


Figure 6. Algorithm Running Time on different Host Number

states without considering the transitional states caused by SDN updates.

Reitblatt *et al.*in the seminal work [22] provide the first formal foundation for network updates through the abstractions of per-packet and per-flow consistency in SDNs and propose a two-phase method to enforce such consistency abstractions. It also reveals security vulnerabilities in SDN due to inconsistent updates. The scope of our work is show that the abstractions proposed in [22] are not sufficient to address many common security and reliability needs. We propose a new complementary abstraction for network updates and our solution naturally guarantees the per-packet consistency.

Noyes *et al.* [19] proposed a tool for synthesizing network updates automatically while satisfying a specified collection of invariants during the transition. The constraints can be specified as linear temporal logic formulas (LTL) formulas. While this approach is more generic and can in theory support a range of inter-flow constraints, it is very expensive – updates synthesis is in the order of ten minutes versus hundred milliseconds in our case. Further, that work does not specifically focus on inter-flow constraints but rather on basic reachability constraints such loop freedom.

McGeer [17] proposed an update protocol that provides per-packet consistency and a weak form of per-flow consistency with the additional goal of conserving switch rulespace. Our proposed update approach for version isolation is very similar to the update approach proposed in [17] albeit with very different goals. An important distinction is that we reinject the original packets back into the network at the source whereas the approach in [17] sends the stored packets directly to the destinations. That approach of sending directly to the destination may break application functionality, especially related to security. For instance, if there was a requirement that certain flows should be routed through a middlebox such as a firewall than this requirement would be violated.

Katta *et al.* [8] point out the high space overhead in the two-phase method and propose to divide the update schedule into multiple rounds. Similarly, the solution in our work also consists of many rounds of updates. Ghorbani and Caesar [4] and zUpdate [14] present update methods under bandwidth constraints. The resource nodes in our dependency graphs provide a similar congestion-free guarantee.

Mahajan *et al.* [5], [7], [15]. Mahajan and Wattenhofer [15] highlight the dependency among rules at different switches and develop an algorithm for loop-free guarantees. Jin *et al.* [7] first presents a solution for dynamic update scheduling and builds a concrete system with dependency graphs. However, these works fail to consider the relationships among different flows during updates. Ghorbani and Godfrey [5] point out the insufficiency for per-packet and per-flow update consistency abstractions and argue for newer update abstractions to account for end application level semantics. Our proposed inter-flow consistency provides a framework to account for end application semantics in a generic way. Our version isolation consistency in particular addresses the needs of some of the applications identified in [5]. However, the coverage and limitations of the proposed update abstraction in terms of addressing the needs of various application classes need to be further explored.

VI. Conclusion

Network updates may take a network into a transitional state that consists of a mix of old and new network configurations. Such transitional states may result in inconsistency of forwarding behaviors and consequently lead to security vulnerabilities. In this paper we argue that existing network update abstractions for SDNs are not sufficient to meet security and reliability requirements that impose constraints across multiple flows. To address this, we argue for a novel update abstraction, inter-flow consistency, that accounts for relationships and constraints among flows during network updates. We focus on two specific types of inter-flow consistency, *spatial isolation* and *version isolation*. To enforce the two isolation properties, we build a dynamic update scheduling algorithm with dependency graphs. We also implement a prototype system for spatial isolation on top of a Mininet OpenFlow network and undertake a preliminary performance evaluation. Preliminary experimental results show that our algorithm has good performance and linear growth over the parameter range tested.

In the immediate future, we are going to implement the part of version isolation in our system. Furthermore, we will study the complexity of our scheduling algorithms and test our solution in a test-bed with real switches as well as conduct large-scale simulation to evaluate the performance of our

system. While spatial and version isolation provide enough flexibility to cover a wide-range of inter-flow constraints, their coverage and limitations need to be further studied and other kinds of inter-flow update consistency abstractions need to be explored.

ACKNOWLEDGMENTS

The authors would like to thank Rhett Smith and Syed Faisal Hasan for valuable discussions. Thanks are also due to the anonymous reviewers for their valuable feedback.

The material in this paper is based upon work supported in part by the Department of Energy[3] under Award Number DE-OE0000679 and by the Air Force Research Laboratory and the Air Force Office of Scientific Research, under agreement number FA8750-11-2-0084.

REFERENCES

[1] Ryu controller. http://osrg.github.io/ryu/. Accessed: 2014-11-01.

[2] Ehab Al-Shaer, Wilfredo Marrero, Adel El-Atawy, and Khalid El-Badawi. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *Network Protocols, 2009. ICNP 2009. 17th IEEE International Conference on*, pages 123–132. IEEE, 2009.

[3] David Erickson, Glen Gibb, Brandon Heller, David Underhill, Jad Naous, Guido Appenzeller, Guru Parulkar, Nick McKeown, Mendel Rosenblum, Monica Lam, et al. A demonstration of virtual machine mobility in an openflow network, 2008.

[4] Soudeh Ghorbani and Matthew Caesar. Walk the line: consistent network updates with bandwidth guarantees. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 67–72. ACM, 2012.

[5] Soudeh Ghorbani and Brighten Godfrey. Towards correct network virtualization. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2014.

[6] Soudeh Ghorbani, Cole Schlesinger, Matthew Monaco, Eric Keller, Matthew Caesar, Jennifer Rexford, and David Walker. Transparent, live migration of a software-defined network. Technical report, Technical report, CS UIUC, 2013. www. cs. illinois. edu/˜ ghorban2/papers/lime.

[7] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 539–550. ACM, 2014.

[8] Naga Praveen Katta, Jennifer Rexford, and David Walker. Incremental consistent updates. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 49–54. ACM, 2013.

[9] Peyman Kazemian, Michael Chan, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *NSDI*, pages 99–111, 2013.

[10] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, pages 113–126, 2012.

[11] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Godfrey. Veriflow: Verifying network-wide invariants in real time. *ACM SIGCOMM Computer Communication Review*, 42(4):467–472, 2012.

[12] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[13] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldmann. Logically centralized?: state distribution trade-offs in software defined networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 1–6. ACM, 2012.

[14] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zupdate: updating data center networks with zero loss. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 411–422. ACM, 2013.

[15] Ratul Mahajan and Roger Wattenhofer. On consistent updates in software defined networks. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, page 20. ACM, 2013.

[16] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. *ACM SIGCOMM Computer Communication Review*, 41(4):290–301, 2011.

[17] Rick McGeer. A safe, efficient update protocol for OpenFlow networks. In *Proceedings of HotSDN*, 2012.

[18] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[19] Andrew Noyes, Todd War, Pavol Černý, and Nate Foster. Toward synthesis of network updates. In *Proceedings of Workshop on Synthesis (SYNT)*, July 2013.

[20] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. Simple-fying middlebox policy enforcement using sdn. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 27–38. ACM, 2013.

[21] Saqib Raza, Yuanbo Zhu, and Chen-Nee Chuah. Graceful network state migrations. *IEEE/ACM Transactions on Networking (TON)*, 19(4):1097–1110, 2011.

[22] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 323–334. ACM, 2012.

[3]Disclaimer: Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.