SFC-Checker: Checking the Correct Forwarding Behavior of Service Function Chaining

Brendan Tschaen*, Ying Zhang†, Theo Benson*, Sujata Banerjee†, Jeongkeun Lee‡, Joon-Myung Kang†
*Duke University †HP Labs ‡Barefoot Networks

***Abstract*—Network middleboxes are difficult to manage and troubleshoot, due to their proprietary monolithic design. Moving towards Network Functions Virtualization (NFV), virtualized middlebox appliances can be more flexibly instantiated and dynamically chained, making troubleshooting even more difficult. To guarantee carrier-grade availability and minimize outages, operators need ways to automatically verify that the deployed network and middlebox configurations obey higher level network policies. In this paper, we first define and identify the key challenges for checking the correct forwarding behavior of Service Function Chains (SFC). We then design and develop a network diagnosis framework that aids network administrators in verifying the correctness of SFC policy enforcement. Our prototype - SFC-Checker can verify stateful service chains efficiently, by analyzing the switches' forwarding rules and the middleboxes' stateful forwarding behavior. Built on top of the network function models we proposed, we develop a diagnosis algorithm that is able to check the stateful forwarding behavior of a chain of network service functions.**

I. Introduction

Network Functions Virtualization (NFV) is a significant transformation of Telco infrastructure to reduce both CAPEX and OPEX while maintaining high carrier-grade service levels. The move to virtualized Network Functions (NFs) on standard servers raises the possibility of reduced performance and increased number of errors and outages. Hence troubleshooting and diagnosing problems early on before deployment is a critical issue. One killer requirement of NFV is service function chaining (SFC), where traffic is steered through a sequence of NFs dynamically. Even with today's physical NFs, constructing a service chain involves multiple components: defining policy, programming the SDN controller, installing switch flow tables, and configuring the NFs. Mistakes in any of these components may cause packets being forwarded to the wrong NFs, or in the wrong order, or dropped. With the emergence of NFV, the scale and dynamics of chaining virtual NFs (VNFs) is likely to increase significantly – these errors will only become more prevalent. Hence, verifying and troubleshooting SFC has become increasingly crucial to the success of NFV adoption. Our long term goal is to build a comprehensive NFV diagnosis and troubleshooting framework into which network, NF and SFC troubleshooting tools can be plugged into, for both static and dynamic, proactive and reactive fault diagnosis and help operational efficiency while maintaining the required SLAs.

Towards the above high level goal, in this paper, we develop a SFC troubleshooting and diagnosis tool. More specifically, we examine whether flows are forwarded correctly according to the high level service chaining policies. We call it *checking or diagnosing the **forwarding behavior** of an SFC*. It includes three aspects, as demonstrated via three illustrative examples in Figure 1. First, it should check the sequence of NFs any flow should traverse. In Figure 1(a), the policy requires all the HTTP traffic from Campus A to Storage servers should be handled by a NAT, followed by a firewall, and finally an IDS. To check the correct implementation of this policy, we not only need to check the forwarding rules on the switches, but also to check how the NFs forward packets. Second, the service chain may be altered dynamically in run time according to the states of the traversed NF. In Figure 1(b), the flow initially goes through an IDS and a Load balancer (LB). If the IDS detects an attack signature in the flow, it alters the service chain to include a firewall for policy enforcement, e.g., dropping the malicious traffic. We call it dynamic service chains, and we aim at checking its correct implementation in both switches and NFs. Besides checking a network path, we also check NF configurations. The third example in Figure 1(c) illustrates the difficulties of performing such static analysis in the presence of NFs. The policy specifies that a Web request from Contractor to the Payroll server should be blocked by the FW. Checking this policy in a path without NFs is comparatively easy: check the rules on the FW and see if it blocks the right range of source IP addresses. However, it is hard when a NAT hides the original source IP addresses before the FW. Note that we only handle the forwarding behavior of an SFC. NFs' other non-forwarding related behaviors, such as counting, traffic optimization, are out of the scope of this paper. We plan to address these extensions in future work.

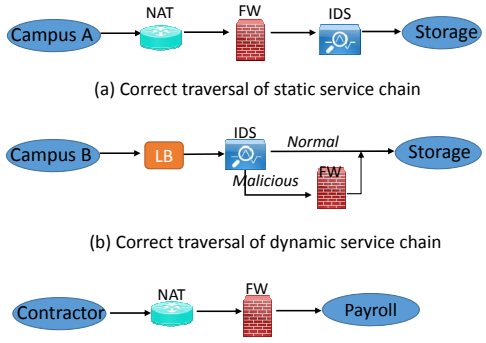

Fig. 1. Example queries

One straightforward way to catch errors in the SFC forwarding is to monitor the flows at run time and then compare

978-1-5090-0933-6/16/$31.00 © 2016 IEEE

the observed path with the policy. However, by the time the error is detected, traffic has already been affected. In this work, we argue for a static analysis framework to capture the problems *before* deployment. This is often known as "network verification" in the SDN context [1], [2]. Different from formal code verification, network verification tools essentially examine rules on all the switches in the network.

Our goal follows the same spirit to perform "verification" on SFC forwarding behaviors. However, we found that existing methods cannot be directly applied to check SFC correctness for a few reasons. First, a policy is complex and stateful. For example, a policy can specify that unauthorized users are prevented from accessing sensitive servers. To do this, an operator could use a stateful firewall to ensure that only traffic initiated from within the network is permitted and in doing so protect users from malicious traffic. NFs maintain each flow's states and perform different actions based on these states. Second, existing forwarding abstractions (e.g., Openflow) cannot be directly applied because all packets of a flow are handled the same using a match-action rule. Thus, we need a new forwarding abstraction to consider the disparate state for individual flows. Finally, to check an SFC, we must check all NFs and switches that the flows traverses: essentially, verifying the entire network. While verifying stateless network devices is computationally challenging [1], [2], adding stateful devices further complicates the problem.

To address these challenges, we make the following contributions. First, we leverage existing middlebox abstract models and generalize them to a *forwarding model* for NF data planes. Each NF is described using a flow table and a state machine. We extend existing Openflow based match-action rules in two ways: (1) the match condition includes not only packet headers but also NFs' internal states, and (2) the action consists of not only modifying packets but also triggering NF state transitions. Further, the match and action can be defined against either an individual packet or on the entire flow. The temporal relationship between states are described using a Finite State Machine (FSM). Note that modeling every detail of a complex NF is clearly intractable without applying sophisticated code analysis techniques. Instead, we focus only on the *forwarding* behavior of the NF. Different from recent extensions to make Openflow stateful [3], [4], [5], our model is specific to NFs and the SFC checking problem. Second, we develop an efficient algorithm for the static analysis of stateful networks. Existing approaches build forwarding graphs from the rules and verify using these graphs [1], [6]. However, this approach is insufficient because the forwarding graphs only capture the forwarding behavior of the network, but not the state transitions of the NFs. We propose a Stateful Forwarding Graph (SFG) that encodes both the state transitions and forwarding behavior. We develop an algorithm that automatically generates SFGs from NF tables and FSMs. Additionally, we designed several graph traversal algorithms on the SFG that answers state-dependent reachability questions.

In this work, we design and implement SFC-Checker, a framework that performs correctness checking of forwarding

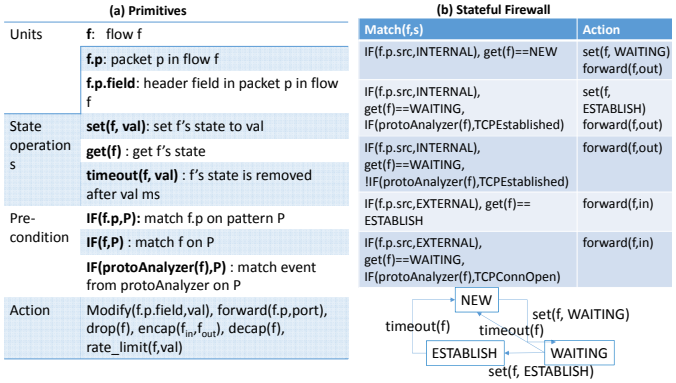

Fig. 2. Model primitives and example.

behavior for service function chains. Similar to existing verification work [1], [6], [7], we focus on checking stateful reachability invariants: e.g., "Given a specific packet trace, what sequence of NFs will the flow traverse", or "After what sequence of packets, will A be allowed to communicate with B?". We developed a prototype of SFC-Checker and used it to analyze SFC diagnosis speed and scalability. The preliminary evaluation results show that we can verify an NF with 27 states in 1s and a network of 1800 NFs per path in 12ms. The time to construct the stateful forwarding graph is around 100ms - 300ms. For the NFs in the evaluation, we developed the forwarding models manually. A future effort will focus on automatic model generation.

II. NF Forwarding Abstraction

NFs can transform the packets in a complex manner. For example, a NAT will modify the source IP and source port. A WAN optimizer may terminate the TCP connection from the client and start a new one with the server. Many commercial NFs have a combination of complex features, e.g., a Bluecoat firewall [8] can also act as a web proxy, an IDS, and a router. Creating a model that captures every detail of an NF is challenging. However, leveraging prior work, it is possible to create a model for the *forwarding behavior* of an NF.

A. Existing modeling methods

Due to their proprietary nature, diverse functionality, and high complexity, it is extremely challenging to model the behavior of a middlebox. Nearly all existing work generates an abstract model to describe an NF's high level behavior. Table I summarizes the most recent work. They are constructed either based on expert domain knowledge or manual investigation of the source code. We summarize them according to the representation of their match fields, stateful representation, and whether the actions are defined against on the packet or on the state. Existing network verification tools [2] and network controller [12] employ simplified middlebox models that do not capture state or packet sequence. Although different in the modeling granularity and format, we found that all of the models can be converted to the form we describe below, which is suitable for efficient diagnosis purposes.

B. NF Forwarding Model

It is the simple switch forwarding abstraction, i.e., Openflow, that makes the data plane verification feasible [1] as it

TABLE I
TAXONOMY OF MIDDLEBOX MODELS.

		MM [9]	VIP [7]	SymNet [10]	BUZZ [11]	HSA [2]	Pyretic [12]	SFC-Checker
How to obtain	Source code	x	x	x	✓	x	x	x
	Expert knowledge	✓	✓	✓	x	✓	✓	✓
Match fields	L2/L3 Header	✓	✓	✓	✓	✓	✓	✓
	L4-7 payload	✓	✓	✓	✓	x	x	✓
State representation	State	✓	x	✓	✓	x	x	✓
	Packet sequence	x	✓	x	x	x	x	✓
Action	On packets	✓	x	✓	✓	✓	✓	✓
	On states	✓	✓	✓	✓	x	x	✓

hides control plane complexity and represents a unified interface. Inspired by the Openflow match-action abstraction, we propose a new NF forwarding abstraction. This NF abstraction is focused on just the forwarding behavior of the NF, and while similar to prior models has three important differences: (1) our model is particularly suited to our stateful diagnosis framework, (2) we model state transition events at arbitrary granularity - from the packet level all the way to events such as connection establishment, which we exploit to contain state space explosion and (3) our model is promising for modeling more complex middleboxes with arbitrary internal packet processing functions. Finally, existing NF models can be easily converted to our model.

Our NF abstraction comprises of two parts: a match-action table and a state machine. The state machine is a natural representation of stateful processes. The nodes in the state machine are the states each NF maintains related to the forwarding behavior, and the edges capture the conditions that trigger state transitions. The table contains match-action rules: matching on both the packet header and the internal states, performing the action on packets, and changing the internal states.

We chose this model for two reasons. First, we found that all existing models in Table I can be converted to the form of Finate State Machine (FSMs). It means that this representation is general enough to capture a variety of middleboxes. Second, we found that existing efficient verification methods are based on graph traversal algorithms. The FSM+Table model of middleboxes can be easily incorporated in existing graph data structures. Such a representation is important for developing scalable and efficient diagnosis algorithms on top of it.

The proposed abstraction can be constructed either from analyzing the source code [11] or converted from existing high level middlebox models [9], [13]. In this paper, we do not focus on the automatic creation of the abstract model. Instead, given such an NF forwarding model, we focus on the SFC checking/diagnosis algorithms.

Abstraction primitives: For illustration, we convert one existing Middlebox Model [9] to the form of our forwarding abstraction with three major changes: 1) separating the state machine and the match-action model; 2) defining the rules against both packets and flows, a.k.a., packet sequence; 3) considering a larger set of actions.

Figure 2(a) shows the set of primitives we currently support. But other primitives can be easily incorporated. The *Units* of a rule can be a field, a packet or a flow, as we may have different rules for packets in the same flow. The *State operation* includes *get(f)* to retrieve the NF's internal state for flow f, and *set(f, val)* to initiate/modify the NF's internal state for flow f to value val. The *Precondition* and *Action* can be defined based on packets and flows. Moreover, since NFs may parse application headers from packet payloads, we further define a $protoAnalyzer$ primitive, which assumes that the NF follows layer 5-7 protocol specifications and generates the protocol specific events, e.g., HTTP request, FTP request.

Examples: Using the primitives, we create models for six NFs: NAT, load balancer, stateful firewall, IDS, VPN gateway, and PDN gateway. These NFs have been heavily studied, and their forwarding behavior can be modeled using this approach. They cover 5 out of the 8 common middlebox types according to a recent survey middlebox usage [14]. For the sake of brevity, we illustrate using one example in Figure 2(b) of a stateful firewall, which maintains three connection states. We use the TCP $protoAnalyzer$ primitive to abstract the stateful behavior. The TCP $protoAnalyzer$ outputs TCP related events such as "Connection Open" or "Session Established". The rules specify that an external packet will only be forwarded if it is a SYN/ACK of a connection initiated by internal hosts, or from an established connection.

III. STATEFUL REACHABILITY ANALYSIS ALGORITHM

Our static analysis of the forwarding behavior occurs in three stages: we first take a snapshot of the network state, i.e., the topology, flow tables and the NF model; Second, using this state, we create a *Stateful Forwarding Graph (SFG)*, which represents how packets are forwarded and how NF states are changed accordingly; and finally, we develop graph traversal algorithms on the SFG to answer various queries to aid operators in their SFC diagnosis tasks.

A. Stateful Forwarding Graph

If all devices are stateless, we can create a stateless forwarding graph that represents the topology and the routing paths similar to [1]. With stateful NFs, we create an SFG by composing the forwarding graph with the NFs' state machines. The composition process is a special case of lexicographic graph composition; the vertices of one NF's state machine is merged only with the vertex corresponding to the NF in the stateless forwarding graph. An SFG captures how packets are forwarded between devices, how states are changed within devices, and how the state changes affect the forwarding path.

In an SFG, each node is denoted as $< H, D, S >$ representing any packet in the packet header space H arriving at a network device (switch or NF) D, when the network device is in a particular state S. An edge pointing from one node $< H_1, D_1, S_1 >$ to another $< H_2, D_2, S_2 >$ means when a

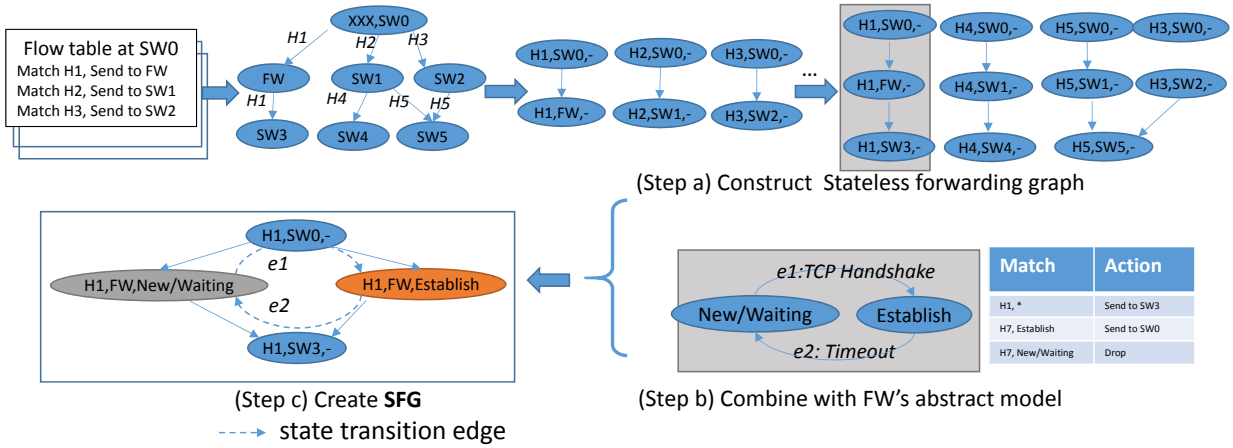

Fig. 3. Stateful Forwarding Graph and the construction process

packet in H_1 arrives at D_1 with state S_1, it will be modified to H_2 and forwarded to a device D_2 at state S_2. In Figure 3 (c) packets H_1 traverse through SW_0, FW, and SW_3. Essentially, there are two types of SFG edges:

Forwarding edge: A forwarding edge from $< H_1, D_1, S_1 >$ to $< H_2, D_2, S_2 >$ means that D_1 modifies packet H_1 to H_2 and forwards the modified packet to D_2. We visualize these edges as solid edges in our examples (Figure 3(c)). All edges in a stateless forwarding graph are forwarding edges.

State transition edge: In a stateful NF, a state transition can be triggered by the previous packets of a flow. Thus, packets of the same flow may be treated differently, depending on its internal states. We represent it using a state transition edge, meaning an NF's state is modified because of the current packet traversal through the state. Each state transition edge is associated with a transition condition, e.g., next packet, or HTTP reply, which identifies the event when the transition will happen. It is shown as a dotted edge e_1 in Figure 3(c). A state transition edge is always between the same device. Similarly, another state transition edge e_2 is added, reflecting FW changing back to $New/Waiting$.

In many NFs, flows in the forward and reverse directions share the same internal state. Thus a state transition in one direction should be appropriately reflected in the other direction. To do this, we create state transition edges between the NFs' state in both directions. For example in Figure 4, the outgoing flow (H_1) goes from SW_0 to SW_3 through the stateful firewall FW while the incoming flow (H_7) takes the reverse path. FW will only forward packets H_2 if it has seen the outgoing packets before. We create an edge from $< H_1, FW, N/W >$ to $< H_7, FW, E >$. It is labeled with e_1 because the state transition is triggered by the same event: the TCP handshake initiated by H_1.

B. SFG construction

One naive way to construct an SFG is to enumerate the path and states each packet traverses. Clearly, this approach is not scalable, as the packet header space can be huge and thus the size of the SFG is not tractable. To overcome this challenge, we leverage the insight that many flows traverse the same set of NFs and switches, which can be grouped together as an equivalent class. A flow is a set of packets sharing some common header fields, e.g. 5 tuples. Second, many flows will encounter the same action when they are in the same state (e.g. connection established), which can also be grouped together. Based on these two intuitions, we group flows into equivalent classes. Similar to [1], [2], an equivalent class is defined as a set of flows sharing the same action on all the network devices at any states. The flows in the same equivalent class traverse the same SFC and have the same actions. Flows sharing the same action at a particular device will be represented by just one node in an SFG.

We propose an SFG construction algorithm below. It starts with a large aggregated flow space, e.g., $< XXX, SW_0 >$ in Figure 3 (a). For simplicity, we use stateless devices here, so the states are all "-". Then, we can get the forwarding actions for the aggregated flow space from the rules. The flow table in Figure 3 suggests the three next hops. We create the next hop nodes according to the matching conditions. When the next hop nodes are created, we split the parent node $< XXX, SW_0 >$ to three nodes $< H_x, SW_0 >, x = 1, 2, 3$. Then we take another node and repeat the same process. When a child node is split, the effect should be propagated back to all its direct parents. This process continues until all rules are parsed. At the end, it generates a stateless forwarding graph.

Next, we take the NFs' forwarding rules and state machines, and expand every stateful NF node in the original graph to many nodes. Each node uniquely represents the NF at a particular state. Using the gray-colored nodes of Figure 3 (a) and (b), we expand nodes with FW to four nodes, representing FW in both forward and reverse direction. The nodes cannot be shared on the forward and reverse directions because FW has different actions. State transition edges are also added according to FW's state machine and the forwarding rules. In the stateful firewall example, because timeout (e_2) can occur on both forward and reverse directions, there are four e_2 edges shown in Figure 4.

C. Verification on SFG

The static analysis leverages the observation that for each flow the NF will be in *one and only one state at any given time*. Essentially, only one node in an NF's state will be active.

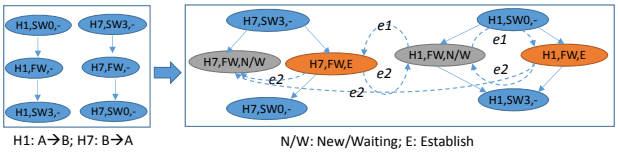

Fig. 4. SFG with forward and reverse directions

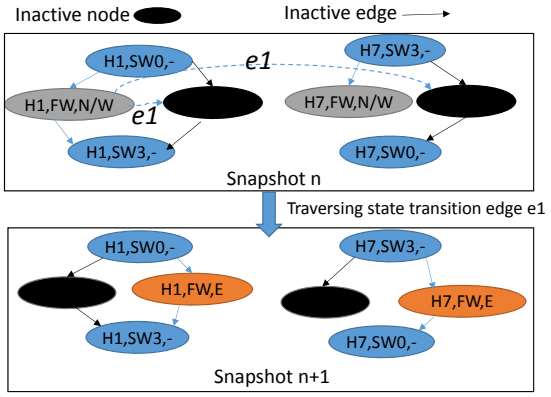

Fig. 5. Temporal forwarding behavior analysis

By activating different nodes (corresponding to different states of an NF) during the static analysis process, we are able to check different forwarding scenarios across NFs and states. To do this, we augment the SFG, such that each stateful NF node (e.g., $< H_1, FW, N/W >$) has a bit to indicate if it is active at the current snapshot of the validation. SFC checking occurs in multiple steps.

Initialization: To initialize the SFG, we create a snapshot of the SFG where-in all NFs are in their initial states. Figure 5 shows an example: at snapshot n, the left branches in both directions are active while the right branches are inactive.

Traversal: Next, given the snapshot of the network, we run a graph traversal algorithm to identify the paths that satisfy the queries, e.g., to answer the Query 1 in §III-E, we search the SFG for all paths from A to B.

Reachability analysis: The existence of such a path determines that A can talk to B but not the conditions for this to occur. To determine the conditions, we need to explore the path and identify the set of state transitions that are required to move the network from $snapshot\_0$ to an alternate $snapshot\_N$ such that all nodes in the path discovered between A and B are activated.

To do this, we search for all stateful nodes in the paths. For each such node, we check to see if there is an incoming dotted state transition edge connecting this node to another stateful node - if so, then we try to find the conditions that must be met to activate this parent node – upon activation both the parent and the original node become active. We note that the process of activating the parent is recursive, as the parent itself may need activation. For example, in Figure 5 checking "H7" the left DAG requires activating the black node. To do this the static analysis algorithm follows the state transition edge to the parent node, $< H_1, FW, N/W >$ and determines that "e1" is required for activation – upon applying "e1", the two black nodes are activated.

```
Verification(source, destination){
  EventList = []
  foreach SFG in SFG List
    If SFG:root == source
      verify(SFG, destination)
}
```

```
Verify(SFG, dst){
  EventList = []
  Paths = BreadthFirstSearch(SFG.root, dst)
  foreach path in Paths
    foreach node in path
      If node:isNotActive
        EventList.add(SFG.Activate(SFG, node))
  return EventList }
```

```
Activate(SFG, node){
  EventList = []
  foreach ancestor in node:getParents
    If ancestor:ContainsStateEventTransitionTo(node)
      EventList.add(ancestor.getStateTransitionTo(node))
      EventList.add(Verify(SFG, ancestor))
  return EventList
}
```

Fig. 6. Pseudocode for SFC traversal: Query 1.

To activate nodes, the checker applies the appropriate event to the SFG and in-doing so creates a new snapshot of the network. When activating a state node for FW, the checker must ensure that other state nodes for FW are deactivated. This process of applying an event to the current snapshot and of activating and deactivating nodes creates a new snapshot. For example, Figure 5, we apply event "e1" to snapshot N, which turns off the gray nodes and activates the orange nodes.

Example: We use Figure 4 as an example to walk through the algorithm. Assuming packets from A to B are in the header space H_7, we first use a breadth first search to find the two paths that H_7 traverses. Next, we find out the path that leads A to B, which contains nodes $< H_1, SW_3, - >$, $< H_1, FW, E >$, $< H_1, SW_0, - >$. In order for this path to be active, we find that there is an incoming dotted edge e_1 for node $< H_1, FW, E >$. We add e_1 to the $EventList$, meaning that in order for this path to be active, e_1 should occur. After we traverse all the paths and all the nodes with incoming dotted edge, we return the $EventList$ link list.

Algorithm scalability: Let's assume a network with n paths and the average path length of l. There are m number of NFs on each path, each of which has k states. Then the size of the SFG is $n(l + km)$.

D. Optimizing algorithm efficiency.

Event ordering: As discussed earlier, during reachability analysis, the process of activating a node requires exploring and potentially recursively activating other nodes. The order of traversal determines the number of nodes we need to recursively activate – this in turn impacts the runtime of our algorithm. We use domain specific knowledge, to determine which parent node has a smaller set of potential dependencies. For example, a parent node requiring a 'SYN' packet event has a smaller than dependency than a parent node requiring a 'HTTP Request' – as the latter requires three packets (e.g. 'SYN', 'SYN-ACK', 'ACK+Request').

Symbolic representation: An NF may modify the packet header fields to a range value where the exact value will only be selected in the run time. For example, a NAT will assign a port to a flow in the run time, so in the static analysis, we will only know the port range. In this case, we use a symbolic variable to represent the modified port. The variable is associated with a value range. The reachability analysis process checks against each possible value. This approach helps reduce the number of nodes in SFG.

E. Multiple queries

We have developed algorithms to support four different queries. Similar questions have been examined in the stateless network context [2]. We address similar queries with stateful extension. Due to the limited space, we only show the Pseudocode of the first query.

- *Query 1: Under what scenarios, can all A's packets reach B?* The algorithm for this query is shown in Figure 6. It uses a breadth first search at each SFG snapshot and modify the graph using the *Activate* function for each event.
- *Query 2: Given a packet/event sequence, can A talk to B?* It injects the packets and events (e.g. timeouts) one by one, uses a depth first search to find the path, and activate the states if current packet triggers the transition.
- *Query 3: What are the service chains that a packet sequence will traverse?* It injects packet sequence, runs a depth first search at each snapshot and records all the paths. It reports multiple service chains if the chain is altered in the middle of the flow.
- *Query 4: Will there be any flows and packet sequences that cause NF X's state m and NF Y's state n coexist?* The reachability analysis runs a breadth first search to find all the paths traversing any of these two states and identify if there is any overlap.

IV. Prototype And Evaluation

We developed a prototype of SFC-Checker in approximately 2279 lines of Java. Our prototype takes as input routing rules from Mininet [15] to create a network topology. We evaluated the time complexity of SFC-Checker on both a linear topology and on a k-level single-rooted tree topology with total 2^k switches. To evaluate its scalability, we implement a generic NF that we can change the number of states. We also evaluate it on a testbed with four types of real NFs.

Metrics: To understand the scalability of SFC-Checker, we evaluated it on two metrics – SFG construction time and SFC checking time (SFG traversal time). We analyzed two dimensions: 1) the number of states per NF and 2) number of stateful NFs per service chain.

SFG construction time: Figure 7 shows the time to construct the SFG with different topology size on a linear topology (top) and with more complex NFs (bottom). Overall, the construction time is small: for a network of 200 nodes (each node has 1 state) the time is 100 ms. The resultant SFG contains 11,938 nodes and consumes approximately 45MB memory. When the number of states increases to 30, in a 10 node topology, the SFG construction time is 310ms. Both experiments have small variance.

SFC checking time: Figure 8 plots the average checking time over 100 different runs as the topology size increases for the four queries in § III. The reachability analysis time depends on the actual service chain length. Query 4 is not shown as each node has only one state in this experiment. Query 1 is most expensive as it searches all the possible paths and states. The time is from 2ms to 12 ms. Figure 9 shows that the checking time for Query 1 and 4 are clearly correlated with the number of states in each NF but not for the other two queries. In Query 1, the checking time increases from 8 ms to 1 s while we increase per-NF states from 2 to 27. When the per-NF states grow to 52, it takes 20s to answer Query 1 and 0.2s for Query 4. In most cases, the number of states per NF per flow is below 20 [16], which means sub-second checking time. Together with the 10ms SFG construction time, we demonstrate the scalability of SFC-Checker on a realistic setting. We also experiment with 4-NF service chain on a tree topology with 11K-node SFG and the checking time is less than 200ms. This speed benefit comes from 1) combining flows and states to equivalent classes; 2) the SFG construction algorithm iteratively splitting nodes top-down, instead of combining huge number of nodes from bottom-up; 3) the checking algorithm processing the events ordered by the domain knowledge.

TABLE II
Results of different SFC implementation.

SFC Implementation	Avg. rules	SFG construction	SFC checking
ODL SFC L2 [17]	3102	7.3ms	14.9ms
ODL SFC NSH [18]	2716	5.5ms	14.8ms
ContexNet [19]	10241	7.9ms	15.1ms

Testbed evaluation: We evaluated SFC-Checker on ContextNet [19], a commercial SFC system used in mobile networks trials and in many industry forums [20]. It emulates the Gi-LAN topology with 4 switches and 15 NFs including Parental controls, video optimization, HTTP header enrichment, big data analytics, and firewall. It uses Openflow and OpenDayLight controller. We generate the models for these 15 NFs by parsing their configurations, and extract the Openflow rules as input to SFC-Checker. The SFC-Checker is able to check the reachability in 15ms. The SFG is built in 8 ms with a size of 112 nodes.

Supporting different SFC implementation: We have tested SFC-Checker with various industry-grade SFC implementations to show its applicability in practice. The results are shown in Table II. Different implementation results in different number of rules in the switches and thus has slight impact on the SFG construction time but not on the SFC checking time.

V. Related work

Middlebox modeling Our NF abstraction are inspired by a long line of research on understanding and modeling NFs [9], [21], [7], [16], [22], [23], [24], [25], [26]. In particular, [22], [16] also models the internal states of NFs but they are designed for migration purposes. Our work focuses on the NF forwarding abstraction for the purpose of forwarding analysis.

Stateful SDN Recently there are proposals to make SDN interface programmable and stateful. [4] proposes to add simple states to Openflow and [5] proposes a stateful programmable data plane. SFA [27] proposes a hardware implementation of the stateful forwarding abstraction and it was fully evaluated in SDPA [3]. While these work focus on programmability and realization of such a stateful interface, SFC-Checker explicitly models the transition between states, differentiates actions on packets versus flows, and illustrates using real NFs.

Middlebox troubleshooting Our work is closely related to BUZZ [11], which builds the FSM from NFs' source code and then generates testing packets based on the FSM. Our work is

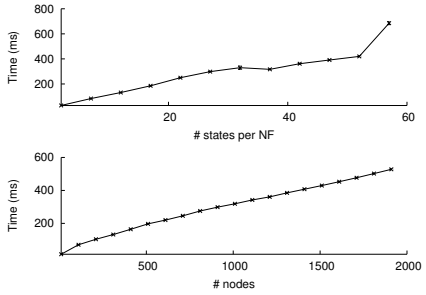

Fig. 7. SFG construction time.

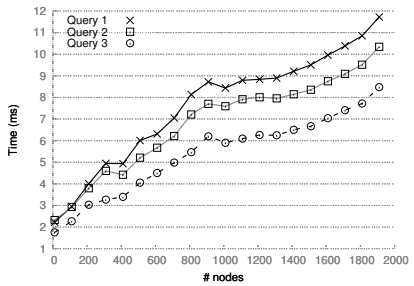

Fig. 8. SFC checking time (diff. network sizes).

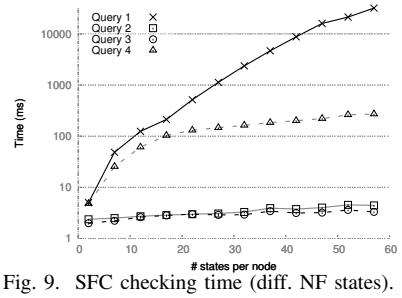

Fig. 9. SFC checking time (diff. NF states).

orthogonal as we focus on network-wide reachability analysis instead of each individual NF. While the FSM constructed from BUZZ can also be used in SFC-Checker, it can also take the high-level NF models and verify the correctness of configurations. The most recent work [7] proposes a modeling language and uses a SAT solver to check the isolation properties. The scalability of this approach is limited by the SAT solver. It does not explicitly support those stateful queries that SFC-Checker verifies. Other network verification [2], [1], [6], [28] focus solely on layer 2 and layer 3 network devices – ignoring stateful devices, e.g., NFs. The recent improvements of SymNet [29] uses symbolic execution on the stateful middleboxes, but it requires writing the NFs using their SEFL language.

VI. Conclusion

In this paper, we have proposed SFC-Checker, a network diagnosis framework that checks correctness of forwarding behaviors of service function chaining in the network. We first propose a simple abstract model that captures the high level states an NF maintains. The action of an NF depends on the packet header and its internal states. Using the model, we propose an algorithm that checks the network forwarding behaviors under different dynamic scenarios: different packet sequences and its resulting NF states. SFC-Checker employs an algorithm to efficiently combine equivalent flow space and state space. While the initial results show promises, we plan to validate applicability of our abstract model to more real NFs and use them to further evaluate SFC-Checker. While the static configuration is useful to check configuration errors, we also plan to extend this work to real-time verification by pulling states from the NFs reactively.

References

[1] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proc. NSDI*, 2013.

[2] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. NSDI*, 2012.

[3] S. Zhu, J. Bi, C. Sun, and C. Wu, "Sdpa: Enhancing stateful forwarding for software-defined networking," in *Proc. International Conference on Network Protocols*, 2015.

[4] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, "Flow-level state transition as a new switch primitive for sdn," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, 2014.

[5] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: Programming platform-independent stateful openflow applications inside the switch," *SIGCOMM Comput. Commun. Rev.*, vol. 44, April 2014.

[6] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, 2013.

[7] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, "Verifying isolation properties in the presence of middleboxes." http://arxiv.org/abs/1409.7687.

[8] "Bluecoat." https://bto.bluecoat.com.

[9] D. Joseph and I. Stoica, "Modeling middleboxes," *Netwrk. Mag. of Global Internetwkg.*, 2008.

[10] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Symnet: Static checking for stateful networks," in *Proceedings of the 2013 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMiddlebox '13, 2013.

[11] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar, "Buzz: Testing context-dependent policies in stateful networks," in *Proc. NSDI*, 2016.

[12] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular SDN Programming with Pyretic," *USENIX*, vol. 38, October 2013.

[13] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, "Verifying isolation properties in the presence of middleboxes." Technical report: arXiv preprint:1409.7687,2014.

[14] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: network processing as a cloud service," in *Proc. ACM SIGCOMM*, pp. 13–24, 2012.

[15] "Mininet: An Instant Virtual Network on your Laptop." mininet.org/.

[16] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, 2014.

[17] "The Service Function Chaining (SFC) OpenFlow Renderer (SFCOFL2)." https://github.com/opendaylight/sfc/tree/master/sfc-py/sfc/nsh.

[18] " OpenDaylight sfc gerrit project." https://github.com/opendaylight/docs/blob/stable/beryllium/manuals/user-guide/src/main/asciidoc/sfc/odl-sfcofl2-user.adoc.

[19] H. ConteXtream, "Introduction to ContextNet." http://www8.hp.com/h20195/v2/GetPDF.aspx/c04725726.pdf.

[20] A. Noy and G. Mainzer, "Carrier Use Cases With OpenDaylight." OpenDayLight Summit 2015.

[21] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang, "Pga: Using graphs to express and automatically reconcile network policies," in *Proc. ACM SIGCOMM*, 2015.

[22] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013.

[23] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet, "Revealing middlebox interference with tracebox," in *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, 2013.

[24] M. Dischinger, M. Marcon, S. Guha, K. P. Gummadi, R. Mahajan, and S. Saroiu, "Glasnost: Enabling end users to detect traffic differentiation," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, 2010.

[25] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "Snap: Stateful network-wide abstractions for packet processing," in *SIGCOMM*, 2016.

[26] J. McClurg, H. Hojjat, N. Foster, and P. Černỳ, "Event-driven network programming," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 369–385, ACM, 2016.

[27] S. Zhu, J. Bi, and C. Sun, "Sfa: Stateful forwarding abstraction in sdn data plane," in *Presented as part of the Open Networking Summit 2014 (ONS 2014)*, 2014.

[28] W. Zhou, D. Jin, J. Croft, M. Caesar, and P. B. Godfrey, "Enforcing customizable consistency properties in software-defined networks," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, 2015.

[29] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Symnet: Scalable symbolic execution for modern networks," in *Proc. ACM SIGCOMM*, 2016.