2015 IEEE 23rd International Conference on Network Protocols

SDPA: Enhancing Stateful Forwarding for Software-Defined Networking

Shuyong Zhu[1], Jun Bi[1], Chen Sun[1], Chenghui Wu[1], Hongxin Hu[2]
[1]Institute for Network Sciences and Cyberspace, Tsinghua University
[1]Department of Computer Science, Tsinghua University
[1]Tsinghua National Laboratory for Information Science and Technology (TNList)
[2]Clemson University
zhu-sy11@mails.tsinghua.edu.cn, junbi@tsinghua.edu.cn
{c-sun14, wuch13}@mails.tsinghua.edu.cn, hongxih@clemson.edu

***Abstract*—As the prevailing technique of Software-Defined Networking (SDN), OpenFlow introduces significant programmability, granularity and flexibility for many network applications to effectively manage and process network flows. However, OpenFlow only provides a simple "match-action" paradigm and lacks the function of stateful forwarding for SDN data plane, which limits it to support advanced network applications. Heavily relying on SDN controllers for all state maintenance incurs both scalability and performance issues. In this paper, we propose a novel Stateful Data Plane Architecture (SDPA) for SDN data plane. A co-processing unit, Forwarding Processor (FP), is designed for SDN switches to manage state information through new instructions and state tables. We design and implement an extended OpenFlow protocol to implement the communication between the controller and FP. To demonstrate the practicality and feasibility of our approach, we implement both software and hardware prototypes of SDPA switches, and develop a sample network function chain with stateful firewall, DNS reflection attack defense and NAT applications in one SDPA-based switch. Experimental results show that the SDPA architecture can effectively improve the forwarding efficiency with manageable processing overhead for those applications that need stateful forwarding in SDN-based networks.**

I. Introduction

Software Defined Networking (SDN) is an emerging network architecture that provides unprecedented programmability, automation, and network control by decoupling the control plane and the data plane. In SDN architecture, network intelligence and state are logically centralized, and the underlying network infrastructure is abstracted for network applications. As a representative technique of SDN, OpenFlow [19] introduces a "match-action" paradigm for the SDN data plane where programmers could specify a flow through a header matching rule along with processing actions applied to matched packets. OpenFlow switches remain simple and are only in charge of forwarding packets according to flow tables issued by the controller, while all the intelligence is placed at the controller side.

In traditional networks, network functions, such as firewalls, WAN optimizers, and load-balancers, are generally implemented by on-path or off-path proprietary appliances or *middleboxes*. However, middleboxes usually lack a general programming interface, and their versatility and flexibility are also poor [23], [24]. One primary goal of SDN is to enable a network controller to run various network applications and manage the entire network directly by configuring packet-handling mechanisms in underlying devices. Although the programmability of OpenFlow significantly helps manage and process network flows and is effective for many applications on top of the controller, the OpenFlow's simple "match-action" abstraction also brings great challenges in building key network services, such as stateful firewalls, which require advanced packet handling. First, OpenFlow focuses solely on L2/L3 network transport. Its data plane only provides limited support for stateful packet processing and is unable to monitor flow states without the involvement of the controller [26]. OpenFlow may impliedly support partial stateful forwarding in the data plane through instructions, but it still lacks the capability to actively *maintain* state information in the data plane. Even though the recent OpenFlow switch specification introduces OpenFlow pipeline, which contains multiple flow tables, in the data plane, the lack of state-relevant tables and primitives preserves the incapability of supporting advanced stateful network applications. On the other hand, heavily relying on the controller to maintain all packet states could give rise to both scalability and performance issues due to associated processing delay and the control channel bottleneck between the controller and switches [17], [27]. Second, OpenFlow targets fixed-function switches that recognize a predetermined set of header fields and processes packets using a small set of predefined actions. The header fields and actions cannot be extended flexibly to meet diverse application requirements. The limited expressivity of OpenFlow compromises the programmability and capability of the SDN data plane [8], [10].

To address the above-mentioned challenges and requirements, we introduce an innovative Stateful Data Plane Abstraction (SDPA) to enable stateful processing in SDN data plane. In contrast to the simple "match-action" paradigm of OpenFlow, we propose a new "match-state-action" paradigm for SDN data plane. In this paradigm, state information can be maintained in SDN data plane without the heavy involvement of SDN controllers. we propose a generic hardware switch design, which is based on a Stateful Data Plane Abstraction (SDPA) paradigm. A variety of complicated network functions, such as stateful firewall and DNS reflection attack defense,

1092-1648/15 $31.00 © 2015 IEEE
DOI 10.1109/ICNP.2015.45

can be implemented in this hardware platform. The rules in data plane devices can be efficiently enforced by specially optimized data structure and hardware, which can especially support hardware network function chains.

The paper makes the following contributions:

- We propose a novel stateful data plane architecture, SDPA, which supports a new "match-state-action" paradigm in the SDN data plane. This architecture has the generality to support various network applications that need to process state information in the data plane.
- We design and implement an extended OpenFlow protocol to support SDPA. Through this protocol, the SDN controller can communicate with the state processing module FP in switches to manipulate the state information in the data plane.
- We implement both software and hardware prototypes of SDPA switches, and develop a sample network function chain composed of stateful firewall, DNS reflection attack defense and NAT applications in an SDPA-based switch.
- We evaluate our approach with extensive experiments. Experimental results show that the SDPA architecture can tremendously reduce the forwarding latency with manageable processing overhead for dealing with stateful forwarding in SDN-based networks.

The rest of this paper is organized as follows. We overview the concept of state and the SDPA paradigm in II. The detailed design of SDPA is articulated in Section III. We present the implementation of SDPA switch in Section IV followed by the evaluations in Section V. We summarize related work in Section VI. We conclude this paper along with our future work in Section VII.

II. Problem Statement and SDPA paradigm

A. Problem Statement

The term "state" in networking is defined as historical information that needs to be stored as input of processing of future packets in the same flow. In this section, we elaborate what state exactly is through two network functions such as stateful firewall and load balancing. A stateful firewall is a type of firewall that keeps track of the state of network connections and determines packet handling according to associated state information [22]. The states of TCP connections and UDP pseudo connections are maintained in the state table, where a state table entry is created when a connection is established. Then, when a packet comes in, the firewall matches the packet to the state table information to determine whether it is a part of a legitimate communication session. If the packet matches a current table entry and obeys state transition policy of TCP/UDP protocol, it is allowed to pass through the firewall. Some other network functions, such as load balancing, also need to maintain state information for packet processing. An important issue when operating a load balancing service is how to handle information that must be kept across the multiple requests in a user's session. The records of ongoing TCP connections between clients and servers are the state information that need to be maintained. In the current SDN architecture, switches can not keep session data.

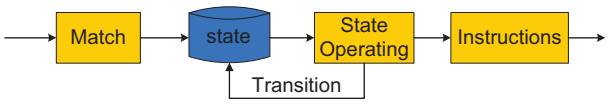


Fig. 1: SDPA paradigm

In summary, the following reasons make it necessary to realize stateful processing in the data plane. Firstly, the state information of each packet in some applications needs to be recorded for advanced packet handling. In such a situation, if the state information is maintained in the controller, there will be considerable Packet-Ins sent to the controller. The forwarding efficiency of SDN would be significantly affected, because it causes significant forwarding latency and the bottleneck between controllers and data plane. Secondly, existing SDN techniques only provide limited support for stateful processing in the data plane. OpenFlow's simple "match-action" paradigm is almost stateless [26]. Under such a circumstance, it is challenging to fully support those advanced stateful network functions in SDN-based networks. Thirdly, although some of advanced network functions can be implemented with middleboxes, middleboxes usually lack a general programming interface [14], [23]. The network is filled with various middleboxes and the structure of the network becomes complex. Consequently, it is critical to design a systematic mechanism for supporting efficient stateful processing in SDN data plane.

B. SDPA Paradigm

Although OpenFlow's "match-action" paradigm is simple and capable enough to support many data plane functions, it provides limited support for stateful processing due to the lack of state-related modules in the pipeline of OpenFlow date plane. In essence, the limited "match-action" paradigm seems to be an involuntary outcome of being amenable to high-performance and low-cost implementations, without taking into account a rich set of complicated network functions (such as stateful firewalls, load balancing, FTP, NAT, etc).

We propose a new "match-state-action" paradigm for SDN data plane as shown in Fig. 1. In this paradigm, we add state fields and state operating instruction to enable stateful processing in SDN data plane. The state fields are used to keep state information of flows or packets and the state operating instructions are used to maintain state information. It is a general paradigm and can be implemented through a diversity of hardware platforms, such as CPU, NPU, NetFPGA, ASIC, etc. When implementing stateful network applications such as stateful firewalls, input packets are processed according to related state information. Then, the state information is updated according to incoming packets or internal/external events. With this new paradigm, state processing can be programmed by SDN applications and the state information can be maintained in SDN data plane. Thus, based on this paradigm stateful processing can be efficiently supported in the data plane without conveying all packets to the controller for state information maintaining.

In stateful SDN data plane, the inputs can be divided into two categories: one is incoming packets, the other is the states

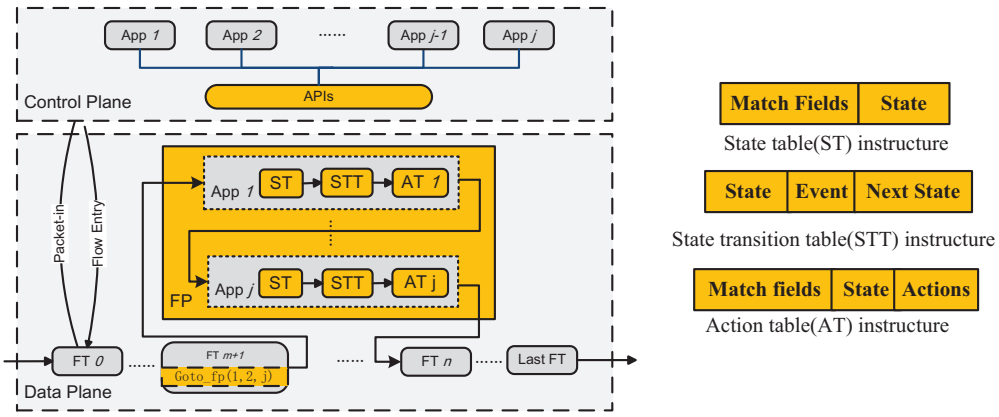


Fig. 2: SDPA architecture

of flows or packets. They are under the control of the transfer function. The outputs include both packets and states. We define S as nonempty finite set of states in SDN data plane. Σ is defined as the input packet set of SDN data plane. We define A as a collection of actions for the data plane, including forward, modify, drop, etc. Δ is defined as a transfer matrix issued by the controller. s_0 is defined as the start state. It is the state when a switch has not processed any input packets. F is defined as the set of final states. F is a subset of S. Then the data plane can be abstracted as a five-tuple model $(S, \Sigma, \Delta, s_0, F)$.

In the paradigm of traditional SDN data plane, the transition matrix can be expressed as formula 1. In this paradigm, the input of this transition matrix Δ is the input packet set Σ alone and the output is simply the output packet set Σ.

$$\Delta : \Sigma \xrightarrow{A} \Sigma \tag{1}$$

In SDPA paradigm, the input packet set Σ and state set S in the data plane converts to the output packet set Σ and state set S. The transition matrix can be expressed as formula 2.

$$\Delta : \Sigma \times S \xrightarrow{A} \Sigma \times S \tag{2}$$

In this new paradigm, the state information of flows or packets is maintained in the data plane. Stateful processing can be smoothly supported. The paradigm is a generic SDN data plane processing paradigm.We developed prototype systems based on the software and hardware in this work, which will be detailed below.

III. Design

In OpenFlow architecture, packets are simply forwarded based on flow tables in switches. Through adding intelligence to switches, they can maintain state information in the data plane in SDPA architecture. Concretely, we design a co-processing unit in SDN switches named Forwarding Processor (FP), which can be implemented using CPU, NPU, NetFPG, etc. Through extended OpenFlow instructions, flows or packets are directed from the OpenFlow pipeline to FP. FP realizes more complex processing of flows or packets through instructions. We design state tables for FP, in which FP maintains the associated state of flows or packets. Fig. 2 depicts the design of SDPA architecture.

A. Forwarding Processor

FP maintains the state of flows or packets. Besides, it can modify the metadata of packets, and initiate or delate state table entries asynchronously. FP can receive and react to incoming events from both the controller and the switch, such as configuration change, state change, or just packets.

We add a GOTO_ST(n) instruction in the data plane, which is used to direct packets from OpenFlow pipeline to state table n in FP. And the packet is directed from FP back to the flow table m through the instruction GOTO_FT(m). We design instructions for stateful processing in FP. These instructions can be flexibly extended to meet application requirements in the data plane. The instructions can be divided into following categories: Control instructions: they are used to direct packets transferring between the controller, flow tables and FP, including GOTO_ST(n), GOTO_FT(m). Processing instructions: they are used for FP to process flows or packets. State operating instructions: they are used to operate the state table. Arithmetic instructions: they are used to perform arithmetic operations. Logical instructions: they are used to perform logical operations.

B. Extended OpenFlow Protocol

The controller and FP communicate with each other through an extended OpenFlow protocol. It is mainly used for the operations of state information in the data plane. such as initialization of the state table and state transition table, action table, etc. The controller has a full control of FP. We design two new message types, *controller-to-FP messages* and *asynchronous messages* for the new protocol. Each of them contains multiple sub-types. *Controller-to-FP messages* are initiated by the controller and used to manage or inspect the state of the FP. Note that *controller-to-FP messages* may or may not require a response from the FP. *Asynchronous messages* are initiated by the FP and used to update the controller of changes to state information. They are sent without a controller soliciting them from the FP. The FP sends *asynchronous messages* to the controller to denote the state changes or other events, including STATE_ENTRY_REMOVE and STATE_ENTRY_MODIFY.

STATE_ENTRY_REMOVE messages are triggered when the state table entry is removed because of timeout or other reasons. STATE _ENTRY_MODIFY messages are used for FP to notify the controller for the changes of state table entries.

C. State Manipulation

In order to maintain state information in SDN data plane, we design three kinds of tables: *state table(ST)*, s*state transition table(STT)* and *action table(AT)*.

1) State table(ST): State tables are used to maintain state information in SDN data plane. Since different protocols may need to maintain different state information, each protocol has a corresponding state table. State tables are initiated by the controller dynamically. When an application requires stateful processing, the controller informs FP to initiate corresponding state tables through an extended instruction INIT. The controller tells FP explicitly which domains the state table should have.

The state information is updated according to incoming packets or internal/external events, and maintained in the data plane. The state information can also be uploaded to the controller through the asynchronous messages, so that the controller can keep the *global* state information of the network. When the state information is updated in FP, it can be sent to the controller to retain consistency. The controller can decide how often switches send STATE_ENTRY_MODIFY messages to controller according to application requirements. For example, switches may send STATE_ENTRY_MODIFY messages to controller after a period of time. It doesn't need to sent STATE_ENTRY_MODIFY messages to the controller after every change.

Fig. 2 shows the structure of state tables. The "Match Fields" domain in a state table refers to the match fields of packets. It is flexible and extensible. For example, it can store connections possibly represented by both source and destination addresses. The "State" domain in a state table is used to record the state information of flows or packets. And the "Instructions" domain is utilized to record associated processing instructions to process packets and update the states. Those instructions can be divided into *State Operating* instructions and *Packet Processing* instructions. The realization of state tables can be based on TCAM, SRAM, etc. State table entries are removed from state tables in two ways, either receiving a request of the controller or via the state table expiry mechanism. The controller may actively remove state table entries by sending state operating instructions. We also design an expiry mechanism for state tables.

2) State transition table(STT): We design a state transition table to support the specification of state update policies with respect to a specific connection-oriented protocol. Each state table should be accompanied by a state transition table. A state transition table specifies the transition policies indicating how the states transfer according to the specific protocol. A state transition table contains three different domains, including *State*, *Event*, *Next State*, as shown in Fig. 2. State transition tables are issued to FP by the controller.

Let us consider a case of TCP protocol. The automaton describes the normal behaviors of a TCP connection. The states of TCP connection include LISTEN, CONNECTION REQUEST, CONNECTION ESTABLISH, DATA TRANSFER, CLOSING and CLOSED. They are updated according to the events specified by TCP flags which are contained in the headers of TCP packets. The state transition table content presents two types of information: the next state determined by current state and the input events, and the associated instructions to be executed on packets. When illicit packets come, they can be easily identified through the invalid transitions.

3) Action table(AT): The action table(AT) is used to give the according action of each flow or packet. The structure of AT is shown in Fig. 2. "Match Fields" and "State" domains are the same as the domains in ST. The "Actions" domains describe the corresponding actions.

As a general architecture, SDPA can support a variety of applications. Since different applications should maintain different state information, each application should have a specific state table. The first packet of a flow should be sent to the controller to determine which applications the flow belongs to. Then, a corresponding flow entry is issued by the controller carrying the extended instruction GOTO_ST(n). The parameter n refers to the state table ID that the packet should be sent to. At the same time, corresponding state table domain information and a state transition table is issued to FP for stateful processing in the data plane. It also can issue flow entries proactively before flow arriving.

D. SDPA APIs

In order to support flexibly-defined stateful functions, we design corresponding north bound API on top of the controller and south bound API between the controller and FP. North bound API is mainly used for administrator to program applications, determine its processing logic, and initialize the state table, state transition table and action table. South bound API is mainly used for communication between the controller and FP. The controller initializes state tables, state transition tables and action tables in the FP through the south bound API. There are four major elements needed to be determined in state tables and state transition tables: match fields, state, event and instruction. API design associated with those four elements are elaborated as follows.

1) South bound API: The south bound interface includes the following key components. (1) Match fields: A match field definition describes the sequence and value of a serious of header fields. Match fields can be flexibly defined according to application requirements. Different APPs may have different match fields. We extend current match fields in OpenFlow flow tables by assigning the position and length of the new fields, including TCP flags. (2) State: State can be flexibly defined as an *enum* data structure since switches need not understand the meaning of each state. The controller can construct the state table and the state transition table using the enum value of state and send them to the switch. (3) Event: Event is the trigger of state transition. For instance, the TCP flag carried in each packet triggers the TCP session state transition. We standardize events into $Param1 + Comparison\_Operator + Param2$ form. FP can fetch $Param1$ and $Param2$ from packets, tables, the switch and the controller. If necessary, the two params can come from the same source, such as *Packet*

Source IP Address and *Packet Destination IP Address*. The Comparison operator is restricted to $<$, $>$, $=$, $\geq$ or $\leq$. Events may vary in different APPs. We consider an event is happened if this (in)equation is satisfied, which will trigger a state transition according to relevant STT entry. For instance, if the *TCP flag* of a *packet* = *FIN*, the state of this connection will be triggered from *ESTABLISH* to CLOSING. (4) Instruction: We classify the functions of Instruction into several categories as discussed above. An instruction is defined in $InstructionType + Parameter$ form. Instructions can be flexibly extended as long as we assign their execution methods and necessary parameters in both the control and data plane.

2) North bound API: The north bound API can be divided into three kinds of functions. Table formation function: Users call those APIs to fill table content using previously defined match fields, state, event and instruction. Message construction function: Those functions are used to build messages transmitted between the controller and the switch, including initializing table formation and conducting table modification. Message transmission function: These functions are used to send messages to the switch.

E. SDN Switch Architecture Supporting SDPA

We design an SDN switch architecture supporting SDPA as shown in Fig. 3. We add *FP* and *State Table* to SDN switch architecture to maintain the state information in the data plane. Besides, we add *a policy module*, which is used to adjust the processing policies. This module includes the state transition table discussed above. The new architecture consists of following functional modules:

- Network interface: it is directly connected to the physical layer and its main functions include receiving/sending packets and packet processing. It works in the physical layer and the link layer.
- Forwarding engine: it is responsible for determining the packet forwarding paths. It parses the received packet headers and looks up the forwarding table to obtain the destination ports for the forwarding operation.
- Forwarding processor (FP): it interacts with the controller and is responsible for the maintenance and management of state information in the data plane.
- Forwarding table: it plays the role of connecting the entire system. It can be updated according to the information issued by the controller and returns associated forwarding instructions to the forwarding engine.
- State table: it is used to maintain state information during the processing procedure in the data plane.
- Policy module: it is used to adjust and control the processing policies of the switch such as the state transition policy, packet processing policy, etc. The policies are issued by the controller.

IV. Implementation of SDPA switch

SDPA architecture is a generic architecture that it can be implemented in a variety of ways. To demonstrate the feasibility and efficiency, we implemented both software and hardware

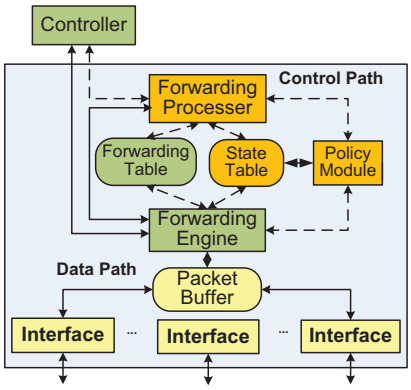


Fig. 3: SDN switch architecture supporting SDPA

Match fields					State				
SIP	SPORT	protocol	DIP	DPORT	Connection state	Sequence number	Acknowledge number	Idle timeout	Hard timeout

Fig. 4: State table structure of stateful firewalls in SDPA architecture

prototype systems of the SDPA switch. And we developed several applications such as stateful firewall, DNS reflection attack defense and NAT to form a network function chain both in SDPA software platform and in hardware platform.

A. SDPA Implementation in Software

In our implementation, we extended Open vSwitch (OVS) [5] to support FP and used Floodlight [6] as the controller, on which we developed three applications including stateful firewall, DNS reflection attack defense and NAT. The SDPA switch runs in Ubuntu 12.04 system running on a DELL OPTIPLEX 780 computer. The CPU of this computer is Intel(R) Core(TM) 2 Duo Processor E7500 (2.93 GHz) and the internal memory is 3.21GB. The network card is Intel(R) 82567LM-3 Gigabit Network Connection. The controller runs on another computer with same configuration. We used IXIA [2] to generate and send original packets in our testbed environment.

We implemented the stateful firewall application based on the SDPA architecture,where FP is used to maintain the state of TCP connections and UDP pseudo connections. The state tables reside in FP to record state information. A detailed structure of state tables in the stateful firewall application is depicted in Fig. 4. The "Match fields" domain consists of *SIP*, *SPORT*, *protocol*, *DIP* and *DPORT*. And the "State" domain contains *Connection state*, *Sequence number*, *Acknowledge number* , *Idle timeout* and *Hard timeout*. The "Instructions" domain includes *state operating instructions* and *packet processing instructions*. When a packet arrives, it is matched against the flow tables to examine if there is a corresponding flow entry. If not, the packet is sent to the controller to match firewall rules through Packet-Ins. If the packet is allowed to pass through the firewall, the controller issues a new flow table entry to the switch, whose instructions contain GOTO_ST(n). Also, the controller issues a new state table entry to FP, whose instructions contain GOTO_FT(m).

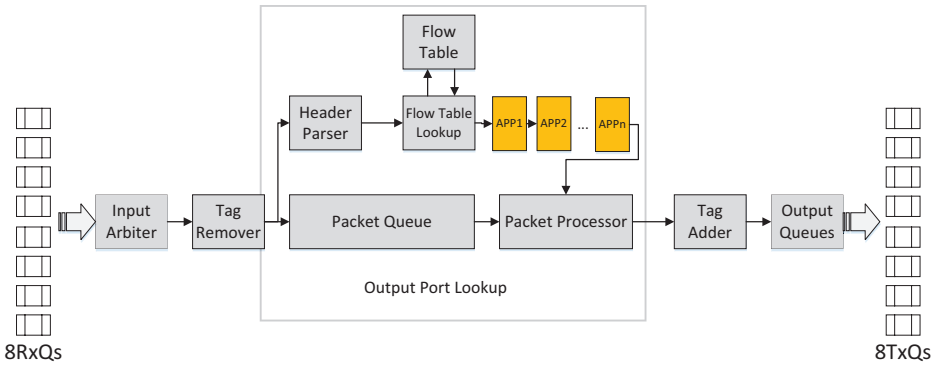


Fig. 5: Hardware packet processing pipeline in SDPA architecture

Algorithm 1: Packet Processing in SDPA-based Stateful Firewalls

Input: Input packets Σ, the state of packets or flows S.
Output: Output packets Σ', the state of packets or flows S'.

foreach $\sigma \in \Sigma$ **do**
  $Flow\_Entry\ e$;
  $e = Switch.Match\_Flow\_Table(\sigma)$;
  if $e = NULL$ **then**
    $Switch.Send\_Packet\_In(\sigma)$;
    ** The packet is matched against firewall policies. **
    if $Controller.Match\_Policy\_Table(\sigma) = Allow$ **then**
      ** The controller issues a new flow entry with instructions $GOTO\_ST(n)$ where n is the ID of state table. **
      $Controller.Issue\_Flow\_Entry(\sigma, GOTO\_ST(n))$;
      ** The controller issues a new state entry with instructions $GOTO\_FT(m)$ where m is the ID of flow table. **
      $Controller.Issue\_State\_Entry(\sigma, GOTO\_FT(m))$;
    else
      $Controller.Issue\_Flow\_Entry(\sigma, Drop)$;
  else if $e.Instruction = Drop$ **then**
    $Switch.Drop\_Packet(\sigma)$;
  else if $e.Instruction = GOTO\_ST(*)$ **then**
    ** The packet is sent to state table n in FP. **
    $Switch.GOTO\_ST(n)$;
    ** The sequence number of a packet is checked to defend against replay attacks. **
    if $FP.Check\_Sequence\_Number(\sigma) = Allow$ **then**
      ** The packet is matched against the state table to check its legitimacy. **
      if $FP.Match\_State\_table(\sigma, n) = Allow$ **then**
        $S' = FP.Update\_State\_Table(\sigma, n)$;
        $\sigma' = FP.Process\_Packet(\sigma, n)$;
        ** The packet is sent to the flow table m. **
        $FP.GOTO\_FT(m)$;
      else
        $FP.Drop\_Packet(\sigma)$;
    else
      $FP.Drop\_Packet(\sigma)$;

Then, the packet is sent to the state table n in FP to maintain the TCP session states or the UDP Pseudo connection states. A state table entry is created when a connection is established through the SDN switch. Every entry of the state table holds a list of information that uniquely identifies the communication session it represents. Subsequent messages are directly sent to FP to match the corresponding state table to determine whether it is a part of a legitimate communication session. According to the current state of the connection and the input event, the associated action and the next state are decided based on the state transition policies defined in the state transition table. The pseudocode of packet processing in stateful firewalls based on the SDPA architecture is given in Algorithm 1.

B. SDPA Implementation in Hardware

To validate the feasibility of SDPA, we also implemented a proof-of-concept hardware prototype using the ONetCard platform [4]. The ONetCard development platform is an acceleration card supporting four Gigabit Ethernet interfaces and two 10G network interfaces based on PCI Express which provides a hardware board similar to NetFPGA. Its center is the FPGA device Kintex7 (XC7K325T-2), which connects network sub-system, storage sub-system, CPU connection subsystem and inter-board sub-system. As the programmable center of the entire ONetCard developing board, the Xilinx Kintex7-325T FPGA provides over 326 thousand logic cells.

We modified the datapath of the OpenFlow hardware switch portion as Fig. 5 depicts. The hardware packet processing pipeline is composed of seven stages: (1) RxQs input queues: buffering packets received from the Ethernet physical ports and DMA virtual ports. (2) Input Arbiter: selecting one input queue through polling and dealing with that queue. (3) Tag Remover: detaching the VLAN tag from original data packet. (4) Output Port Lookup: core module for packet processing inside which the packets are temporarily buffered in the packet queue and the Header Parser analyses the packets to get the header fields. The Flow Table Lookup module and the State Table Lookup module match the header fields in flow tables or state tables and get according instructions. The Packet Processor deals with packets according to the instructions, such as modifying the header fields, dropping the packet or outputting the packet. (5) Tag Adder: combing the processed packet with VLAN tags to form a complete packet. (6) Output Queues: sending the packet to relevant output queues on the basis of the processing decisions of the packet. (7) TxQs Output Queue: buffering the output queue to corresponding

output port. The following two modules are added for stateful processing in ONetCard platform .

1) State table: It is used to store the state information. We use TCAM + SRAM to keep the state table. The match fields of state table can be flexibly defined according to application requirements. For different applications, the fields of state table may be different. Taking TCP protocol as an example, the state table is equipped with a TCAM with 64 entries of 104 bits, which is the length of the five tuple (Source IP, Destination IP, Source Port, Destination Port, Protocol), and an associated block RAM of 64 entries of 8 bits (length of TCP state) that reads the output of the TCAM. Since each of the TCP connections are bidirectional, we use two adjacent table entries to hold the state of one TCP connection. Those two entries are established and updated at the same time, ensuring that packets from both directions can be matched in the state table. With the development of hardware, TCAM capacity will certainly be increased to meet the demand.

2) State table lookup: It realizes the management, lookup and update of the state table. Its input is the packet header and associated TCP flag. The Header Parser will analyze the packet header and look up the state table. The Packet Processer refers to the result of state table lookup, which is denoted by 0 or 1 to decide whether the packet should be allowed to pass or be dropped.

C. Customization of Network Function Chain in an SDPA Hardware Switch

SDPA hardware switches support the customization of network function chains. It offers scalability through the SDPA paradigm and corresponding APIs. In SDPA hardware switches, network functions can be deployed, updated and removed flexibly through configuration from the controller. For instance, to deploy a new network function to the SDPA hardware switch, we just need to configure the state table and the state transition table through APIs from the controller. It can support the deployment of new functions through pre-configuration hardware resources in the SDPA hardware platform. We developed a sample network function chain based on the SDPA hardware switch, which includes stateful firewall, DNS reflection attacks defense and NAT functions.

In a DNS reflection attack, attackers send DNS requests to name servers using the victim host's source IP address, so the victim will be flooded by the name servers' responses. To filter out these unsolicited responses, the SDPA gateway of the victim network maintains the requests sent out from the local network, and checks the validity of the incoming responses. In detail, packets whose UDP source or destination port equals 53 will be sent to the DNS reflection attack defense pipeline. In the ST, an unmatched DNS request will trigger the switch to install 2 new entries. There are four states in the state transition table: the initial state, the state where a request is sent, the state where the connection is legitimately closed, and the state where an unsolicited response is detected.

NAT is a methodology of remapping one IP address space into another by modifying network address information in IP datagram packet headers while they are in transit across a traffic routing device. It includes basic NAT, one-to-many NAT, etc. All these NATs are implemented in a routing device that uses stateful translation tables to map the private addresses into a public IP address and readdresses the outgoing packets so they appear to originate from the routing device. In the reverse communications path, responses are mapped back to the originating IP addresses using the rules stored in the translation tables. In the traditional SDN architecture, the flow tables cannot support the stateful translation tables. While in SDPA architecture, the stateful translation tables can be easily supported. The incoming packets can directly query the corresponding IP addresses in SDPA-based data plane.

D. Dynamic Deployment of New Applications on SDPA Hardware Switch

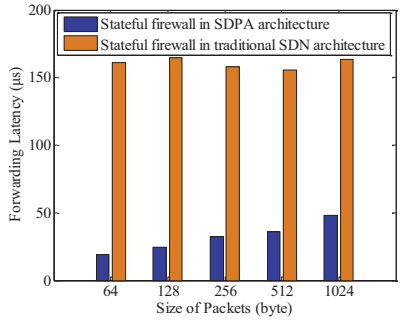
Based on the unified SDPA paradigm and southbound APIs, SDPA hardware switches are more scalable than traditional middleboxes. For those applications that can be abstracted into the SDPA paradigm, SDPA hardware switches support the dynamic deployment of these applications through the pre-configuration of hardware resources. If an application needs to maintain state information in the data plane, and its actions have been developed in SDPA switches in advance, it can be deployed on SDPA switches by the configuration from the controller.

The new application can be deployed in the network function chain in any order according to application requirements. It can be implemented through adjusting the action GOTO_APP(i) of the flow tables and the action NEXT_APP(k) of applications in the network function chain. It is deployed on SDPA hardware switches through the following steps. Firstly, the controller calculates corresponding state table and state transition table of the application. Secondly, the controller sends the encapsulated messages to SDPA switches, which are used to install the state table, the state transition table and the action table of the new application. Thirdly, the software layer of the switch parses the messages issued by the controller and installs the tables into the hardware card. Among them, the time needed for the first step depends on the needs of the new network function and the programmer's efficiency.

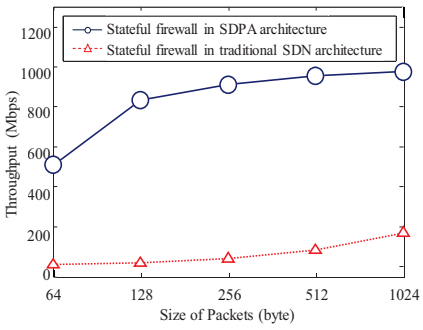
V. EVALUATIONS

1) Performance of stateful firewalls in SDPA architecture against stateful firewalls in traditional SDN architecture: We conducted a contrast experiment to evaluate the efficiency of SDPA. We evaluated the performance of processing states in switches in the SDPA architecture against processing states in the controller in the traditional SDN architecture. We also developed a stateful firewall application based on the traditional SDN architecture, where the state information is maintained in the controller. In this architecture, much more packets should be sent to the controller to check its state information before forwarding.

We tested the forwarding latency and the throughput respectively. We sent 100,000 packets for each packet size ranging from 64 to 1024 bytes. As can be seen from the experiment results, the average forwarding latency reduces significantly in SDPA architecture than that in transitional SDN architecture as shown in Fig. 6(a). In addition, the throughput increases significantly in the SDPA architecture as shown in Fig. 6(b). When realizing stateful firewalls in the traditional

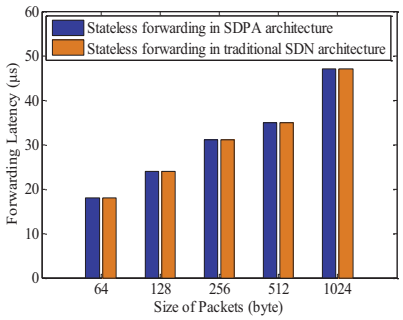


(a) Forwarding Latency

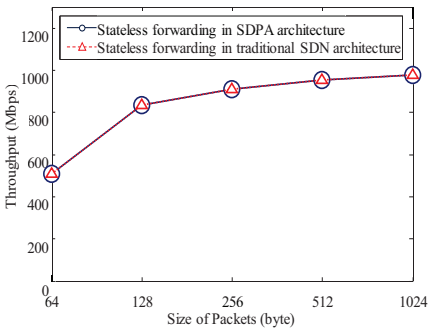


(b) Throughput

Fig. 6: Performance of stateful firewalls in SDPA architecture against stateful firewalls in traditional SDN architecture

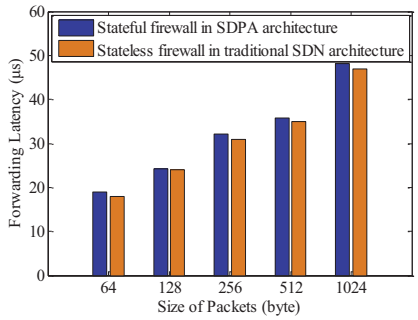


(a) Forwarding Latency

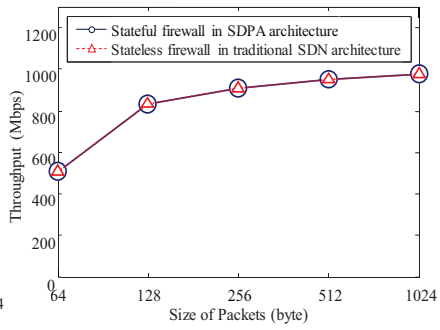


(b) Throughput

Fig. 7: Performance of stateless forwarding in SDPA architecture against in traditional SDN architecture



(a) Forwarding Latency



(b) Throughput

Fig. 8: Performance of stateful firewalls in SDPA architecture against stateless firewalls in traditional SDN architecture

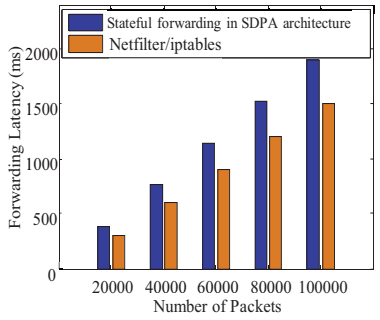
SDN architecture, the processing bottleneck of the controller limits the processing capability of the firewalls. When realizing stateful firewalls in the SDPA architecture, SDN data plane maintains all state information. The throughput of firewalls is significantly improved regardless of the size of packets. The performance improvement of SDPA architecture lies in the architecture, which can maintain state information in data plane that is independent of traffic types.

2) Performance of stateless forwarding in SDPA architecture against in traditional SDN architecture: Since the SDPA architecture is fully compatible with OpenFlow, SDPA can also support *stateless* processing just like OpenFlow. While performing stateless processing in the data plane, the average forwarding latency in the SDPA architecture is almost the same as that in the traditional SDN architecture, as shown in Fig. 7(a). And the throughput in the SDPA architecture is almost the same as that in the traditional SDN architecture as depicted in Fig. 7(b). It demonstrates that the SDPA architecture is fully compatible with OpenFlow. Applications that do not need to maintain state information in the data plane can be fully supported as well without causing additional processing overhead.

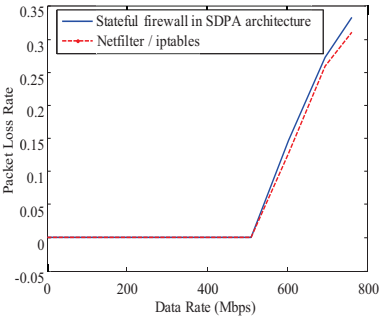
3) Performance of stateful firewalls in SDPA architecture against stateless firewalls in traditional SDN architecture: We compared our stateful firewall in the SDPA architecture with a *stateless* firewall in the traditional SDN architecture. Regarding the stateless firewall in traditional SDN architecture, only the first packet of a flow is sent to the controller to match firewall rules. Then, the controller issues a new flow entry to the flow table. The subsequent packets of the flow are in turn directly matched against the flow table to decide whether those packets should be forwarded or dropped. As can be seen in Fig. 8(a), the average forwarding latency of the stateful firewall in the SDPA architecture is slightly increased. Also, the processing overhead is acceptable and the throughput rate is nearly unchanged as shown in Fig. 8(b).

Considering both the stateful firewall in the SDPA architecture and the stateless firewall in the traditional SDN architecture, when the data transfer rate ranges from 10 Mbps to 980 Mbps, the CPU utilization and memory utilization in the SDPA architecture are almost the same as those in the traditional SDN architecture.

4) Performance of SDPA-based stateful firewall against netfilter/iptables: Netfilter/iptables [3] is a user-space application program that allows a system administrator to configure the tables provided by the Linux kernel firewall and the chains and rules it stores. Nevertheless, our stateful firewall is an application running on top of a controller to enable effective state information processing in SDN-based networks. Although the functions of our stateful firewall application are slightly different to the functions provided by netfilter/iptables, we can still compare their forwarding latency and packet loss rate in the same experiment environment. We selected a gigabit network card and used 64-bytes packet to conduct our experiment. As shown in Fig. 9(a), the forwarding latency of stateful firewalls in the SDPA architecture is just a little bit

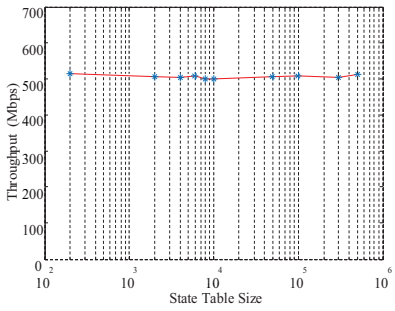


(a) Forwarding Latency

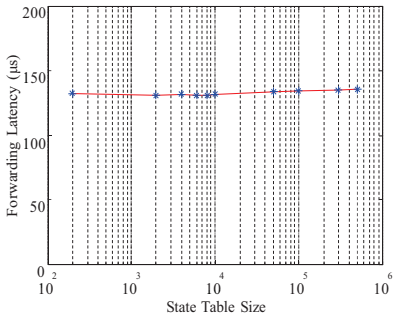


(b) Packet Loss Rate

Fig. 9: Performance of stateful firewalls in SDPA architecture against netfilter/iptables

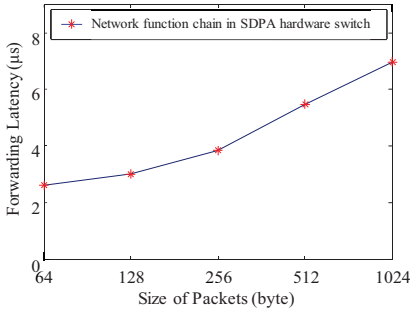


(a) Forwarding Latency

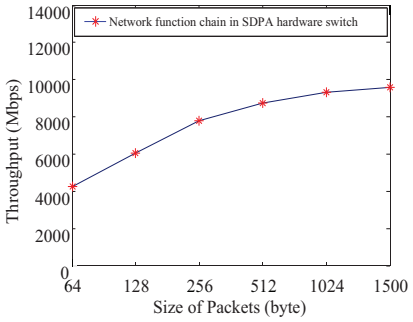


(b) Throughput

Fig. 10: Testing the scalability of state tables



(a) Forwarding Latency



(b) Throughput

Fig. 11: Performance of hardware-based network function chain

higher than that of netfilter/iptables. And the packet loss rates of the two firewalls are almost the same as shown in Fig. 9(b).

5) Testing the scalability of state tables: We performed a test on the scalability of state tables and the influence of forwarding efficiency under different sizes of state tables both in SDPA software prototype and hardware prototype. Since state tables are implemented based on SRAM in software prototype, the size of the state tables can be increased theoretically. We used 64-byte packets to conduct our experiment. As the state table size increases, the forwarding efficiency does not deteriorate obviously as depicted in Fig. 10, in which we use logarithmic scale for x axis. As shown in Fig. 10(a), when the size of state table increases from 200 to 500000, the network forwarding latency does not increase significantly, and the network throughout almost keeps no change as shown in Fig. 10(b). In our experiment, the table look-up process consumes short time and has little variance, exerting little effect on the performance, which will be the only factor as the size of state table grows. The SDPA hardware prototype is developed based on ONetCard, in which state tables are implemented in TCAM. The capacity of state tables can satisfy various application requirements with the increase of the TACM capacity.

6) Evaluation of the network function Chain in SDPA hardware switch: We evaluated the performance of the network function chain implemented on the SDPA hardware switch, which includes stateful firewall, DNS reflection attack defense and NAT functions. We used 10G bits network cards to evaluate the performance of the hardware-based network function chain implemented in the SDPA switch. The experiment results are shown in Fig. 11 . From the experiment results, average forwarding latency is 2.6 μs for 64 bytes size packets, and 6.9 μs for 1024 bytes size packets. Throughput reaches 9566 Mbps when sending 1024 bytes size packets.

7) Evaluation based on a real-world network topology: We also developed a network function chain based on the SDPA software prototype. We performed experiments in a Mininet simulation environment based on a real-world network topology derived from the Stanford backbone network [1]. By using this real world network topology, we attempted to evaluate the efficiency of the network function chain. Each switch in this topology is embedded with the network function chain. We selected a three-hop forwarding path for a TCP connection. The forwarding latency is approximately 95 μs and the throughput is about 9.2 Gbps.

VI. Related Work

Some research efforts have been recently devoted to extend the OpenFlow data plane abstraction [9], [11], [13], [18], [28]. Bosshart et al. [11] pointed out that the rigid table structure of current hardware switches limits the scalability of OpenFlow packet processing to match on a fixed-set of fields and to a small set of actions. They introduced a logical table structure RMT (Reconfigurable Match Table) on top of the existing fixed physical tables and new action primitives. By comparison, we strive to enhance the programmability of the data plane by adding a co-processing unit in SDN switches. Bianchi et al. [9] proposed a new abstraction to

formally describe a desired stateful processing of flows inside SDN data plane based on eXtended Finite State Machines (XFSM). Moshref et al. [20] proposed FAST (Flow-level State Transitions) as a new switch primitive for SDN. Shuyong et al. [28] introduced a preliminary stateful forwarding solution in SDN data plane. However, all of them did not present the relationships and interactions between the state tables and flow tables in SDN switches, thus the compatibility with OpenFlow remains unclarified. They also did not elaborate the fundamental shortcoming of data plane paradigm caused by the incompletion of current OpenFlow specification. Besides, they could not provide concrete implementations and extensive evaluations. In this paper, we presented a novel "match-state-action" paradigm for the SDN data plane and designed an extended OpenFlow protocol for the SDN controller to operate the state information in the data plane. We also developed both software and hardware prototypes based on the SDPA architecture. Especially, we developed three stateful applications and organized them as a network function chain in an SDPA hardware switch, and provided the support of dynamic deployment of new applications.

Since current OpenFlow data plane is limited to support stateful processing, the advanced packet processing has been turned to specialized middleboxes [8], [15]. Anwer et al [8] also believe that expanding the "match-action" interface could make it possible for network operators to implement more sophisticated policies. To support complex middlebox functions in SDN, Fayazbakhsh et al. [12] developed a FlowTags architecture. Such an approach attempts to combine *traditional* middleboxes with the SDN architecture. There are also some efforts for developing middlebox functions using SDN [14], [21], [25]. In particular, Gember et al. [14] advocated for a mechanism that helps exercise unified control over the key factors influencing middlebox operations. Qazi et al. [21] proposed to add an SDN-based policy enforcement layer to efficient middlebox-specific traffic steering. However, those work lacks a general programming interface for applications. Moreover, the network is filled with various middleboxes and the structure of the network is complex. We believe that with SDPA stateful data plan abstraction, new approaches would be stimulated for designing middlebox functions within the SDN architecture. Another option to address current middlebox limitations is to utilize virtualization technologies to manage core networking functions via software as opposed to having to rely on proprietary middleboxes to handle these functions, referred to as Network Functions Virtualization (NFV) [16]. Since SDN and NFV are complementary technologies [29]. We believe our solution can facilitate the realization of stateful network functions in NFV through integrating our SDPA architecture into Service Function Forwarder (SFF) in NFV [7]. Especially, our hardware implementation of SDPA can provide high forwarding capacity to fulfill the requirements of stateful packet processing required by advanced network functions.

VII. Conclusion and Future Work

OpenFlow protocol provides limited support for stateful packet processing in the SDN data plane, which limits it to support advanced network applications. In this paper, we have proposed a novel stateful data plane architecture SDPA. Through adding a co-processing unit FP, it can manipulate state information in the SDN data plane. We have also designed an extended OpenFlow protocol to implement the communication between the controller and the FP. We presented a new "match-state-action" paradigm in the data plane, which has the generality to support various applications that need to process state information in the data plane. In addition, we have implemented both software and hardware prototypes of SDPA switches, and developed a network function chain including stateful firewall, DNS reflection attack defense and NAT functions on a SDPA hardware switch. Our experimental results show that the SDPA architecture can tremendously improve the forwarding efficiency with manageable processing overhead for those applications that need stateful forwarding in SDN-based networks. For the future work, we will develop more stateful network applications based on the SDPA architecture to further validate the effectiveness of our approach. We will also extend the concept of states in our architecture to application-level and customized states to support more comprehensive network applications.

VIII. Acknowledgment

This work is supported by the National High-tech R&D Program ("863" Program) of China(No.2013AA013505) and the National Science Foundation of China (No.61472213). Jun Bi is the corresponding author.

References

[1] Header space library: https://bitbucket.org/peymank/hassel-public.

[2] Ixia: http://www.ixiacom.cn/.

[3] netfilter/iptables project: http://www.netfilter.org/.

[4] Netfpga: http://netfpga.org/.

[5] Open vswitch: http://openvswitch.org/.

[6] Project floodlight: http://www.projectfloodlight.org/floodlight/.

[7] Service Function Chaining (SFC) Architecture: http://tools.ietf.org/pdf/draft-ietf-sfc-architecture-02.pdf.

[8] Bilal Anwer, Theophilus Benson, Nick Feamster, Dave Levin, and Jennifer Rexford. A slick control plane for network middleboxes. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13)*. ACM, 2013.

[9] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. Openstate: programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM Computer Communication Review*, 44(2):44–51, 2014.

[10] Pat Bosshart, Dan Daly, Martin Izzard, Nick McKeown, Jennifer Rexford, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. Programming protocol-independent packet processors. *arXiv preprint arXiv:1312.1719*, 2013.

[11] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM (SIGCOMM'13)*. ACM, 2013.

[12] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *Proceedings of the USENIX Symposium on Networked System Design and Implementation (NSDI'14)*, 2014.

[13] Open Networking Fundation. Software-defined networking: The new norm for networks. *ONF White Paper*, 2012.

[14] Aaron Gember, Robert Grandl, Junaid Khalid, and Aditya Akella. Design and implementation of a framework for software-defined middlebox networking. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 467–468. ACM, 2013.

[15] Glen Gibb, Hongyi Zeng, and Nick McKeown. Initial thoughts on custom network processing via waypoint services. In *3rd Workshop on Infrastructures for Software/Hardware co-design (WISH'11)*, 2011.

[16] R Guerzoni et al. Network functions virtualisation: an introduction, benefits, enablers, challenges and call for action, introductory white paper. In *SDN and OpenFlow World Congress*, 2012.

[17] Michael Jarschel, Simon Oechsner, Daniel Schlosser, Rastin Pries, and Sebastian Goll. Modeling and performance evaluation of an openflow architecture. In *Proceedings of the 23rd International Teletraffic Congress*. International Teletraffic Congress, 2011.

[18] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. Millions of little minions: using packets for low latency network programming and visibility. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 3–14. ACM, 2014.

[19] 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.

[20] Masoud Moshref, Apoorv Bhargava, Adhip Gupta, Minlan Yu, and Ramesh Govindan. Flow-level state transition as a new switch primitive for sdn. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'14)*, 2014.

[21] 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 (SIGCOMM'13)*. ACM, 2013.

[22] Chris Roeckl and Corporate Marketing Director. Stateful inspection firewalls. *Juniper Networks White Paper*, 2004.

[23] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 24–24. USENIX Association, 2012.

[24] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: network processing as a cloud service. *ACM SIGCOMM Computer Communication Review*, 42(4):13–24, 2012.

[25] Seungwon Shin, Phillip A Porras, Vinod Yegneswaran, Martin W Fong, Guofei Gu, and Mabry Tyson. FRESCO: Modular Composable Security Services for Software-Defined Networks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'13)*, 2013.

[26] Haoyu Song. Protocol-Obilivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13)*. ACM, 2013.

[27] Soheil Hassas Yeganeh, Amin Tootoonchian, and Yashar Ganjali. On scalability of software-defined networking. *Communications Magazine, IEEE*, 51(2):136–141, 2013.

[28] Shuyong Zhu, Jun Bi, and Chen Sun. Sfa: Stateful forwarding abstraction in sdn data plane. In *Open Networking Summit 2014 Research Track (ONS'14)*, 2014.

[29] M Zimmerman, D Allan, M Cohn, N Damouny, and Kolias. Openflow-enabled sdn and network functions virtualization. *Solution Brief, ONF, Solution Brief sbsdn-nvf-solution. pdf*, 2014.