SDPA: Toward a Stateful Data Plane in Software-Defined Networking

Chen Sun, Jun Bi, *Senior Member, IEEE*, *ACM*, Haoxian Chen, Hongxin Hu, *Member, IEEE*, *ACM*, Zhilong Zheng, Shuyong Zhu, and Chenghui Wu

***Abstract*— As the prevailing technique of software-defined networking (SDN), open flow introduces significant programmability, granularity, and flexibility for many network applications to effectively manage and process network flows. However, open flow only provides a simple "match-action" paradigm and lacks the functionality of stateful forwarding for the SDN data plane, which limits its ability 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 the 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 open flow protocol to support 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, domain name system (DNS) reflection defense, and heavy hitter detection 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.**

***Index Terms*— SDN, stateful forwarding, data plane.**

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 [20] 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 [24], [25]. A primary goal of SDN is to enable a controller to run various applications and manage the entire network by configuring packet-handling mechanisms in underlying devices. Although OpenFlow significantly helps manage and process network flows and is effective for many applications running on top of the controller, OpenFlow's simple "match-action" abstraction also introduces great challenges in building key network services, such as stateful firewalls, heavy hitter detection, etc., which require advanced packet handling.

On the one hand, OpenFlow focuses solely on L2/L3 network transport. Its data plane provides limited support for stateful packet processing and is unable to monitor flow states without the involvement of the controller [27]. OpenFlow may impliedly support partial stateful forwarding in the data plane through instructions and counters, but it still lacks the capability to actively *maintain* state information in the data plane. For instance, a heavy hitter detection application needs to alert the system when a flow packet counter exceeds a specific threshold. However, in OpenFlow, even if a flow counter can be maintained in the data plane, the flow table cannot react differently when the counter exceeds the threshold. The controller has to pull the counter from switches regularly. But a large pulling interval will result in untimely reaction, while a small interval will consume more network bandwidth. Besides, flow entries are aggregated and usually unable to provide flow-level monitoring, which may not be able to support applications like heavy hitter detection. 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.

Manuscript received December 24, 2015; revised July 28, 2016, October 25, 2016, February 17, 2017, and May 24, 2017; accepted July 5, 2017; approved by IEEE/ACM Transactions on Networking Editor Y. Ganjali. Date of publication August 1, 2017; date of current version December 15, 2017. This work was supported in part by the National Key R&D Program of China under Grant 2017YFB0801701 and in part by the National Science Foundation of China under Grant 61472213. *(Corresponding author: Jun Bi.)*

C. Sun, H. Chen, Z. Zheng, S. Zhu, and C. Wu are with the Institute for Network Sciences and Cyberspace, Tsinghua University, also with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China, and the Tsinghua National Laboratory for Information Science and Technology, Beijing 100084, China (e-mail: c-sun14@mails.tsinghua.edu.cn; chenhx12@mails.tsinghua.edu.cn; zhengzl15@mails.tsinghua.edu.cn; zhu-sy11@mails.tsinghua.edu.cn; wuch13@mails.tsinghua.edu.cn).

J. Bi is with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, also with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China, also with the Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China, and also with CERNET Network Center, Beijing 100084, China (e-mail: junbi@tsinghua.edu.cn).

H. Hu is with the School of Computing, Clemson University, Clemson, SC 29634 USA (e-mail: hongxih@clemson.edu).

Digital Object Identifier 10.1109/TNET.2017.2726550

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 the associated processing delay and the control channel bottleneck between the controller and switches [18], [28]. In addition, OpenFlow targets fixed-function switches that recognize a pre-determined 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 [7], [9].

The limitation of the OpenFlow data plane has been recognized by the research community. Some recent efforts have been devoted to enhance the programmability of the SDN data plane [9], [27]. Among them, P4 [9] is a typical language that allows flexible definition of protocol header fields, parsers, and tables. Although P4 could maintain information in the data plane during runtime based on its *register* data structure, it aims to enhance the data plane programmability. Therefore, it proposes an abstraction for *low-level general data plane behaviors*, without particular focus on the paradigm or abstraction for *high-level stateful applications*. Despite its strong programmability in the data plane, P4 could not support intuitive programming of stateful applications. Furthermore, P4 lacks corresponding control plane that can interact with data plane applications to dynamically issue data plane configurations.

To address the above challenges, we introduce an innovative Stateful Data Plane Architecture (SDPA) to enable intuitive programming and high performance processing of stateful applications in the SDN data plane. In contrast to the simple "match-action" paradigm of OpenFlow, we propose a new "match-state-action" paradigm for the SDN data plane. In this paradigm, state information can be maintained in the data plane without heavy involvement of controllers. Based on the SDPA paradigm, we propose a generic stateful switch design for both software and hardware. A variety of complicated stateful applications, such as stateful firewalls and DNS reflection defense, can be implemented in this platform. The rules in the data plane devices can be configured by the controller and efficiently enforced by specially optimized data structures and stateful processing modules, which can especially support network function chains on software or hardware.

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, short for Forwarding Processor, 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 defense, and heavy hitter detection applications in SDPA-based software and hardware switches.
- We evaluate our approach with extensive experiments. Results show that the SDPA can tremendously reduce the forwarding latency of stateful applications with manageable processing overhead in SDN-based networks.

The rest of this paper is organized as follows. We overview the concept of state and SDPA paradigm in Section II. SDPA design is articulated in Section III. We present the implementation in Section IV, and evaluations in Section V. We summarize related works in Section VI. We present some discussions in Section VII, and conclude this paper in Section VIII.

II. Motivation and SDPA Paradigm

A. Motivation

In this section, we first introduce the stateful firewall application, based on which we elaborate our motivation of maintaining state in the SDN data plane.

The term "state" in networking is defined as historical information that needs to be stored as input for processing of future packets in the same flow. A stateful firewall is a type of firewall that keeps track of the state of network connections and determines packet handling according to the associated state information [23]. The states of TCP connections and UDP pseudo connections are maintained in a state table, where an entry is created when a connection is detected. Then, when a packet comes in, the firewall matches the packet to the state table to determine whether it belongs to a legitimate session. If the packet obeys state transition policies of TCP/UDP protocol, it is allowed to pass through the firewall.

Based on the stateful firewall use case, we summarize the following motivations to maintain state in the SDN data plane. *Firstly*, some applications need to record the state information of *each packet* for advanced handling. If the state is maintained in the controller, there will be considerable Packet-Ins sent to the controller. The forwarding efficiency would be significantly affected because it incurs extra forwarding latency and the bottleneck between the controller and the switches. Moreover, relying on the controller to process state would exert heavy load on the controller and degrade its efficiency. *Secondly*, existing SDN techniques provide limited support for stateful processing in the data plane. OpenFlow's simple "match-action" paradigm is almost stateless [27]. The data plane cannot maintain state and react differently when the state changes. Therefore, it is challenging to fully support stateful applications in SDN. *Thirdly*, although some advanced applications can be implemented in middleboxes, middleboxes usually lack a general programming interface [14], [24]. A network filled with various middleboxes is hard to manage. Consequently, it is critical to design a systematic mechanism for supporting stateful processing in the SDN data plane.

B. SDPA Paradigm

Although OpenFlow's "match-action" paradigm is simple and capable enough to support many applications, it provides limited support for stateful processing due to the lack of state-related modules in the pipeline of OpenFlow data plane. In essence, the limited "match-action" paradigm seems

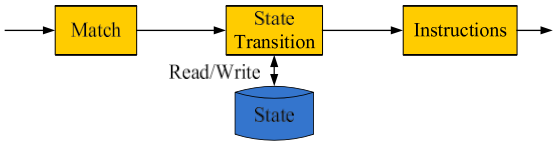


Fig. 1. SDPA paradigm.

to be an involuntary outcome of being amenable to high-performance and low-cost implementations, without considering a rich set of complicated network functions (such as stateful firewalls, load balancing, FTP monitoring, heavy hitter detection, etc.).

We propose a new "match-state-action" paradigm for the SDN data plane as shown in Fig. 1. In this paradigm, we add state fields to keep the state of flows, and extend actions to manipulate state fields. It is a general paradigm and can be implemented through a diversity of software and hardware platforms, such as CPU, NPU, NetFPGA, ASIC, etc. When implementing stateful 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 and the state information can be maintained in the SDN data plane without conveying all packets to the controller for state maintaining.

In stateful SDN data plane, the inputs can be divided into two categories: incoming packets and states of flows. They are under the control of the transfer function. The outputs include both packets and states. We define S as a nonempty finite set of states in the 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 powerset of Σ, which represents both unicast and multicast traffic.

$$\Delta : \Sigma \xrightarrow{A} P(\Sigma) \tag{1}$$

In SDPA paradigm, the input packet set Σ and the state set S in the data plane convert to the output packet set $P(\Sigma)$ and state set S. The transition matrix can be expressed as:

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

In this new paradigm, state information of flows is maintained in the data plane. Stateful processing can be supported with higher performance. Detailed design to support this new paradigm will be elaborated in the following section.

III. Design

In the OpenFlow architecture, packets are simply forwarded based on flow tables in switches. Through adding intelligence to switches, we can maintain state information in the data plane in SDPA. Concretely, we design a co-processing unit in SDN switches named Forwarding Processor (FP), which can be implemented using CPU, NPU, NetFPGA, etc. To reserve the *compatibility* with current OpenFlow architecture, we design extended OpenFlow instructions to direct packets between the flow table and FP. FP realizes more complex processing of flows according to previously introduced *state transition matrixes*, or *state machines* of stateful applications. As depicted in Fig. 2, we design a State Transition Table for FP to describe the state machine of each stateful application. We add State Tables in FP to maintain the state of flows and Action Tables to record actions under different states. Since different applications vary in state transition policies, each application possesses its own table set. For initialization, the controller issues the *entire* state transition table and formats of state table and action table to the data plane. During runtime, the first packet of a flow is sent to the controller to determine which applications should process this flow. Then, a flow entry is issued to direct the flow into FP. At the same time, the controller sends corresponding stateful table entries to FP. The state of this flow is maintained in the data plane. The controller can also issue table entries proactively before flow arrival. We will introduce each module and elaborate SDPA's support for stateful applications in this section.

A. Forwarding Processor

FP maintains the state of flows and processes packets according to the current state and state transition policies. We add a GOTO_ST(n) instruction in the data plane, which is used to direct packets from the OpenFlow pipeline to the state table n in FP. After FP processes the packet, it sends the packet back to the source flow table in the OpenFlow pipeline. In situations where packets are requested to be processed by several applications sequentially inside one switch, the controller can issue several GOTO_ST instructions to direct packets to the corresponding applications respectively.

We design actions for stateful processing in FP. These actions can be flexibly extended to meet application requirements. The actions can be divided into following categories: Control actions: they are used to direct packets between the flow tables and FP, such as GOTO_ST. Processing actions: they are used for FP to process flows, including SET_FIELD, OUTPUT, DROP, etc. State operating actions: they are used to operate the state table, such as STATE_UPDATE. Arithmetic actions: they are used to perform arithmetic operations. Logical actions: they are used to perform logical operations.

B. State Manipulation

In order to maintain state information in the SDN data plane, we design three kinds of tables: *State Table (ST)*, *State Transition Table (STT)*, and *Action Table (AT)*. Since different applications may need to maintain different state information, each application may have a unique corresponding table set.

1) State Table (ST): State tables are used to maintain states in the SDN data plane. STs can be initiated by the controller dynamically. When an application requires stateful processing,

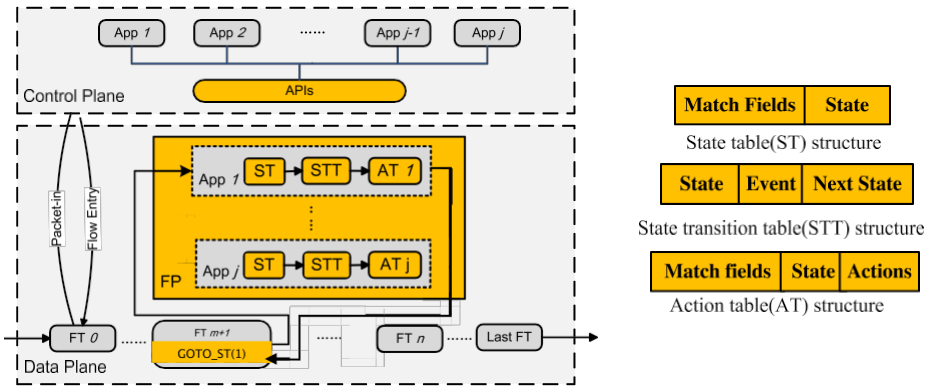


Fig. 2. SDPA architecture and table structures.

the controller sends a message to FP to initiate its ST. The message contains information about all fields of the ST.

The state is maintained in the data plane and updated according to incoming packets or internal/external events. The state information can be uploaded to the controller, so that the controller can keep the *global* state information of the network. The controller can decide how often switches send the update messages according to application requirements. For example, switches may be configured to inform the controller periodically instead of one message for each 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. In accordance with traditional OpenFlow match fields, we also include an equally long match field mask domain in state tables. The "State" domain in a state table is used to explicitly record the state information of flows with no mask. The realization of state tables can be based on TCAM or SRAM. The controller may actively add, modify, or remove state table entries by sending state operating messages.

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. A state transition table specifies the transition policies that indicate how the states transfer according to the protocol. A state transition table contains three different domains, including *State*, *Event*, *Next State*, as shown in Fig. 2. The STT is issued to the FP by the controller *only once* during initialization.

The *State* domain possesses an equally long mask domain, enabling a wildcard match on the current state for different events. The mask is essential for advanced stateful applications. For instance, in the heavy hitter detection, the current counter state should be compared against a threshold, no matter what the counter value is. Without the mask, there will be innumerable table entries.

The *Event* domain is the trigger of state transitions. For instance, the TCP flag carried in each packet triggers the TCP state transition. We standardize events into *Param*1+ *Comparison_Operator* + *Param*2 format. FP can fetch *Param*1 and *Param*2 from packets, tables, the switch, and the controller. The two params can come from the same source, such as *Packet Source IP Address* and *Packet Destination IP Address* from packet headers. The Comparison operator is restricted to $<$, $>$, $=$, or $\geq$. Events may vary in different applications. We judge that an event is detected if this (in)equality is satisfied, which can 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.

The *Next State* domain can be directly and explicitly assigned or calculated through an arithmetic or logical operation. We define the *Next State* domain as *type*+*parameter*. Currently we support two types of *Next State* domain, including DIRECT_ASSIGN, in which case the *Next State* domain is directly assigned by the controller, and ADD_ONE, in which case the *Next State* field will be the result of *State*+*1*.

3) Action Table (AT): The action table (AT) is used to record the actions under different states. 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. We classify the functions of actions into several categories as discussed above. An action is defined in *ActionType* + *Parameter* form. Actions can be flexibly extended as long as we assign their execution methods and necessary parameters in both the control and data plane.

C. SDPA's Support for Stateful Applications

Many network applications can be abstracted into state machines defined over a flow or an aggregation of flows. When a packet enters FP, it first looks up the *ST* for the current state. Using the current state and event (if any), the FP will look up the *STT* and find or calculate a new state. Then the packet and the new state will be sent into the *AT* to be processed accordingly, while the new state will be updated into the *State Table*. Finally, the packet will be sent back to the OpenFlow pipeline for further processing.

We exemplify some stateful applications in TABLE I. We classify the abstraction of stateful applications into two types: *finite state machine* and *infinite state machine*.

1) Finite State Machine: Some stateful applications monitor states that can be abstracted into finite state machines, such as the TCP connection state. Let us consider the case of stateful firewalls that monitor flow TCP states. The states of

TABLE I
OVERHEAD OF TRADITIONAL IMPLEMENTATION AND NEW SDPA ARCHITECTURE FOR STATEFUL APPLICATIONS

	Application Name	Function	Overhead of Traditional Architecture	SDPA Architecture Solution
Finite State Machine	Stateful firewall	Match each packet against TCP finite state machine and filter illicit packets	Controller burden and control channel bottleneck from sending every packet to the controller for state maintenance and inspection	Keep state machine in the data plane and inspect each packet
	DNS reflection defense	Track if a host has sent a DNS request and drop DNS reply packets without a recorded request	Forwarding latency from sending each DNS relevant packet to the controller to record requests and identify legal replies	Track each request and filter the replies without a previous request in the data plane
	FTP monitoring	Only allow inbound FTP data channels set up by FTP control channel	Performance overhead from sending control traffic to the controller to (un)install rules for data channel	Track whether the control channel has been established and filter data plane traffic
	Fast reroute	Output packets through different ports in case of link failure detection	Forwarding latency caused by sending unsuccessfully forwarded packets to the controller	Maintain different output ports for different states (link failures) in the switches
Infinite State Machine	Heavy hitter detection	Detect heavy hitter and send its info to the controller or drop excess traffic	Bandwidth overhead of controller fetching counter state periodically and untimely response for heavy hitters	Maintain a threshold for flows of any granularity and check the counter state in the data plane
	Super-spreader detection	Detect flows from one source with too many TCP connections	Overhead of sending all packets to the controller, since switches cannot monitor the TCP flag field	Maintain a connection number for each source and compare against a threshold in the data plane
	SYN flood detection	Alert when too many SYN packets are flooded through the switch	Overhead of packet-in all TCP packets	Keep a counter of SYN packets and compare against a threshold in the data plane
	ARP spoofing detection	Detect the link of an attacker MAC address and a legitimate IP	Overhead of sending all ARP packets to the controller	Switches keep the MAC-IP mapping state and detect one MAC mapping more than one IPs

State Table:

Match Field				State
Src IP	Dst IP	Src TCP Port	Dst TCP Port	
192.168.2.2	10.0.0.5	8080	50	CLOSED
10.0.0.5	192.168.2.2	50	8080	CLOSED

State Transition Table:

State	Event(TCP Flag)	Next State
CLOSED	SYN	SYN
SYN	SYN + ACK	SYN_ACK
SYN_ACK	ACK	ESTABLISHED
ESTABLISHED	FIN	FIN_WAIT1
ESTABLISHED	NULL	ESTABLISHED
FIN_WAIT1	FIN + ACK	CLOSING1
FIN_WAIT1	FIN	CLOSING0
FIN_WAIT1	ACK	FIN_WAIT2
FIN_WAIT2	FIN	CLOSING1
CLOSING0	ACK	CLOSING1
CLOSING1	ACK	CLOSED
*	RST	CLOSED
*	*	INVALID

Action Table:

Match Field				Next State	Action
Src IP	Dst IP	Src TCP Port	Dst TCP Port		
192.168.2.2	10.0.0.5	8080	50	INVALID	DROP & set state back to current state
10.0.0.5	192.168.2.2	50	8080	INVALID	DROP & set state back to current state
192.168.2.2	10.0.0.5	8080	50	*	NULL
10.0.0.5	192.168.2.2	50	8080	*	NULL

Fig. 3. Table design for stateful firewall.

TCP connection include SYN, SYN_ACK, ESTABLISHED, FIN_WAIT, CLOSING, and CLOSED. They are updated according to the events specified by TCP flags in the headers of TCP packets. Fig. 3 shows the ST, STT, and AT design for the stateful firewall. The *match field* domain in ST and AT are accompanied by masks and can be assigned as wildcards. The *Next State* field in STT is directly and explicitly assigned by the controller. When illicit packets come, they can be easily identified through the invalid transitions and dropped.

State Table:

Match Field	State
Source IP	
10.0.0.1	23
10.0.0.2	192

State Transition Table:

State	Event	Next State
*	State > Threshold	OVER_THRESHOLD
*	*	State + 1

Action Table:

Match Field	State	Action
Src IP		
*	OVER_THRESHOLD	Inform Controller/Drop
*	*	Pass

Fig. 4. Table design for heavy hitter detection.

2) Infinite State Machine: Many network applications need to count packets and react to different counter states, such as heavy hitter detection, flow size monitoring, load balancing, DDoS detection, etc. As for the case of heavy hitter detection, to detect heavy hitters on a server or a subnet with traditional SDN switches, the controller should install counting rules on flow tables, then periodically query the counter statistics from switches. This approach has two major limitations:
(1) Frequent queries from the controller bring severe overhead to the controller. (2) Fetching all of the counter statistics from switches consumes significant bandwidth between controller and switches. Such limitations call for a mechanism to handle state changes without frequent communications between the switches and the controller. Now we introduce the support for heavy hitter detection under the SDPA architecture.

The definition of the table fields is shown in Fig. 4. SDPA uses the *state* field of ST to maintain flow counters. When a packet arrives, we first extract certain fields that identify the flow (e.g. source IP field for heavy hitter detection), then

lookup the counter statistics (*state* field) stored in the ST. The STT is responsible for increasing the counter and checking if the counter is larger than a threshold. We configure a wildcard rule on the *state* field of STT, and the *event* field to compare the current state with the threshold. Here, the *param*1 of event is extracted from the state in ST, and the *param*2 is assigned by the controller. If the counter is larger than the threshold, the state of the flow will be updated as *OVER_THRESHOLD*, otherwise the counter is increased by 1. The AT specifies actions depending on the new state of the flow. If the new state is *OVER_THRESHOLD*, the switch sends the packet to the controller to inform the detection of a heavy hitter. Otherwise, the packet is passed to the next stages.

D. SDPA APIs

In order to support flexibly defined stateful functions, we design north bound APIs on top of the controller and south bound APIs between the controller and the FP. North bound API is mainly used for operators to program applications, which includes the determination of its processing logic and table structures. South bound APIs are mainly used for communication between the controller and the FP. The controller initializes and modifies STs, STTs, and ATs in the FP through the south bound APIs. We elaborate the API design as follows.

1) Key Components in SDPA APIs: The SDPA APIs include the following key components. (1) Match field: A match field definition describes the identification of each flow, such as the five tuples of a TCP connection. Match fields can be flexibly extended according to application requirements. We extend current match fields in OpenFlow flow tables by assigning the position and length of some new fields such as TCP flags. (2) State: A state can be defined as an *enumeration* variable expressed as: *enum state*$\leftarrow\{1, 2, 3, \ldots, n\}$, since switches need not understand the meaning of each state. The controller can construct the state table and the state transition table using the enumeration values of a state and send them to the switches. (3) Event: An event is the trigger of state transition. For instance, the TCP flag carried in each packet triggers the TCP state transition. (4) Action: We specify actions to process packets under different states. Actions can be flexibly extended as long as we assign their execution methods and parameters in both the control and the data planes. Actions supported in SDPA are listed in Section III-A.

2) South Bound API: The controller and FP communicate with each other through the south bound APIs, an extension of traditional OpenFlow protocol. The APIs are mainly used for the operation of state information in the data plane, such as initialization and modification of the table entries from the controller to the FP, and status report from the FP to the controller. The controller has full control over the FP.

With the above four key components, we design two message types, *Controller-to-FP messages* and *asynchronous messages* for SDPA south bound APIs. *Controller-to-FP messages* are initiated by the controller to manage or inspect the state of the FP. They include (1) Table initialization message: this function is used to initialize the tables inside the FP. (2) Entry modification message: They are used to add, modify, or remove state and action table entries. (3) Switch configuration message: They are used for the controller to configure data plane properties, such as state report interval. 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 state changes. 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 the FP to notify the controller for the changes of state table entries.

3) North Bound API: The north bound APIs are provided for operators to program stateful applications. They can be divided into three types of functions. (1) Table formation function: Users call this interface to define table fields including match fields, state, event and action. We provide a fixed set of field choices in current implementation. However, this set can be flexibly extended according to application requirements. (2) Message construction function: This function is used to build messages transmitted between the controller and the switch, including table initialization or modification and switch configuration. (3) Message transmission function: This function is used to transmit messages to the switch.

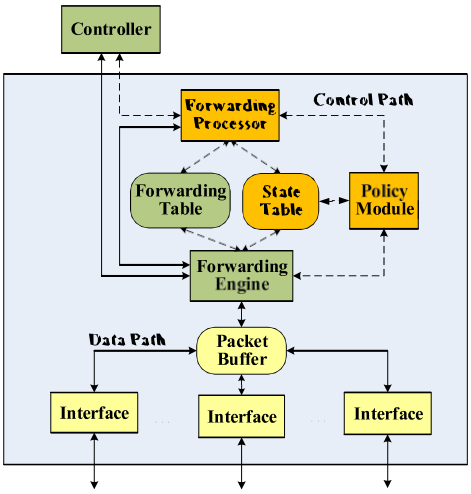


Fig. 5. SDN switch architecture supporting SDPA.

E. SDN Switch Architecture Supporting SDPA

We design an SDN switch architecture supporting SDPA as shown in Fig. 5. 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 STT, AT and state-relevant configurations discussed above. The new architecture consists of the 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 flow tables 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.
- Flow 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 store the processing policies from the controller including the entire STT, AT, configuration of state report interval policy, etc.

F. Controller Enhancement in SDPA Architecture

In traditional SDN architecture, stateful applications can be supported by heavily involving the controller. Each packet has to be sent to the controller for state maintenance. Despite the fact that OpenFlow data plane can support some states, such as counters, it cannot actively maintain state and react to different states. Besides, OpenFlow switches can only track flow counters with the granularity of flow table entries, which are always aggregated to save TCAM resource and cannot meet the requirement of some flow-level stateful applications. Thus, to get flow-level state without wasting data plane resource, packets have to be sent to the controller for state maintenance, which could incur both scalability and performance issues.

In addition to traditional centralized control functions of the controller, in SDPA architecture, we enhance the controller to support stateful applications with higher performance. We design the forwarding processor (FP) inside the switches for state maintenance in the data plane and use SDPA southbound APIs to communicate between the controller and the switches. The controller is able to initialize an application inside a switch. During runtime, the controller can pro-actively add, modify, and delete table entries in FP. Besides, the controller is also able to configure switch properties, including state report interval, etc. From the applications' perspective, the controller exposes north bound APIs for applications to modify ST, STT, and AT to implement stateful processing logics.

The controller can receive the state report from the data plane periodically and update the local record for state synchronization. The switches do not need to inform the controller every time a state transition occurs to save network bandwidth and to relieve controller burden. The state stored inside the controller can be used for failure recovery. The controller can install the state of a failed switch to a new one and redirect flows accordingly. As the controller might not possess the latest state when a switch fails, the state to be installed in the new switch could be inconsistent with the latest state in the old switch. Enabling switch failure recovery with state consistency is beyond the scope of this paper.

Note that the controller still acts as the centralized intelligence in SDPA architecture. Traditional functions such as link discovery, topology detection, forwarding, and so on are still executed by the controller. We add SDPA protocol into the architecture and maintain state in the data plane to enhance both performance and scalability for stateful applications.

IV. Implementation of SDPA Switch

SDPA architecture is a generic architecture that can be implemented in a variety of ways. To demonstrate the feasibility and efficiency, we implemented both software and hardware prototype of the SDPA switch. We also developed several applications such as stateful firewall, DNS reflection defense, and heavy hitter detection to form a network function chain both in SDPA software and hardware platform.

A. SDPA Implementation in Software

We extended Open vSwitch (OVS) [4] to support FP and used Floodlight [1] as the controller, on which we developed three applications including stateful firewall, DNS reflection defense, and heavy hitter detection.

We introduce the workflow of packet processing in the controller, switch, and FP in Workflow 1. This workflow is implemented in both hardware and software implementations of SDPA. 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. The controller issues a new flow table entry to the switch, whose instructions contain GOTO_ST(n). Then, the packet is sent to the state table n in FP to maintain the states. Subsequent messages are directly sent to FP to match the corresponding state table. According to the current state of the connection and the input event, the next state is decided based on the state transition policies defined in the state transition table. Finally, FP looks up the action table to find actions given the next state and the packet match fields. Here, we should note that after action lookup, FP immediately executes the actions, since the processing result of this application might be needed in subsequent stateful applications or OpenFlow flow table entries. Both software and hardware implementations perform in the same way.

According to Workflow 1, we extended OVS to support FP in the data plane. First, we extended the security channel used to communicate with the controller to support SDPA message transmission, including both controller-to-FP messages that initialize and configure stateful applications in switches, and asynchronous messages that report states to the controller. Second, we enriched OpenFlow instructions with a GOTO_ST instruction that directs packets from the OpenFlow pipeline to FP. Finally, we implemented the table lookup and state maintenance logics in FP, in order to support finite or infinite state machines. In the controller side, we also extended Floodlight security channel to transmit SDPA messages. Moreover, we implemented the SDPA north bound API on top of the controller for constructing stateful applications in SDPA.

We implemented the stateful firewall application based on the SDPA architecture where the FP is used to maintain the state of TCP connections and UDP pseudo connections. The ST reside in FP to record state information. A detailed

Workflow 1: Packet Processing in the *Controller, Switch and FP* in SDPA-Based Stateful Applications

Input: Input packets Σ, the state of packets or flows *S*.
Output: Output packets Σ^{l}, the state of packets or flows S^{l}.

```
1  foreach σ ∈ Σ do
2      Flow_Entry e = Switch.Match_Flow_Table(σ);
3      if e = NULL then
4          Switch.Send_Packet_In(σ);
5          /* Example App on the Controller*/
6          /* Install a flow entry. n is the state table ID for
           the app. */
7          Controller.Issue_Flow_Entry(σ, GOTO_ST(n));
8          /* Construct and install corresponding ST and AT
           entries. */
9          Controller.Issue_Entry(ST_Entry, AT_Entry, n);
10     else
11         foreach ins ∈ e.instructions do
12             if ins == GOTO_ST (*) then
13                 /* Switch: Sent packet to state table n in
                   FP. */
14                 Switch.GOTO_ST(n);
15                 /* FP: Match ST, STT, and AT of the app. */
16                 State_Entry = FP.Match_ST(σ, n);
17                 Next_State =
                   FP.Match_STT(State_Entry.State, n);
18                 Actions = FP.Match_AT(σ, Next_State, n);
19                 /* FP: Update the state of the flow in ST. */
20                 FP.Update_ST(Next_State, State_Entry, n);
21                 /* FP: Execution actions from AT to the
                   packet. */
22                 σ^l = FP.Execute_Action(σ, Actions);
23                 /* FP: Packet backs to the original flow
                   table. */
24             else
25                 /* Switch: Original OpenFlow
                   instructions.*/
```

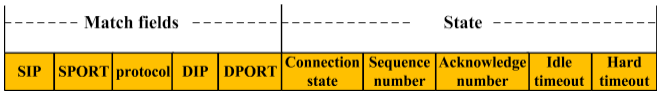


Fig. 6. State table structure of stateful firewalls in SDPA.

structure of ST in the stateful firewall application is depicted in Fig. 6. 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 "Actions" domain includes *state operating actions* and *packet processing actions*.

B. SDPA Implementation in Hardware

To validate the feasibility of SDPA, we implemented a proof-of-concept hardware prototype based on the ONetCard platform [3] . The ONetCard development platform is an acceleration card supporting four Gigabit Ethernet interfaces and two 10G network interfaces based on PCI Express. Its center is the FPGA device Kintex7 (XC7K325T-2), which connects network sub-system, storage sub-system, CPU connection sub-system, 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. The TCAM resource on the board is simulated by Look Up Tables (LUTs) based on RAM on ONetCard platform.

The hardware packet processing pipeline is composed of seven stages as Fig. 7 depicts: (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 gets the header fields. The Flow Table Lookup module matches the packet headers against flow tables to find associated instructions. The Packet Processor deals with the packets according to the instructions, such as modifying the header fields, dropping the packet or setting output ports. (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 output port.

To support SDPA in the data plane, we extended the datapath of the OpenFlow hardware switches as shown in yellow blocks in Fig. 7. We append one stateful processing module for each stateful application. During hardware implementation, we address the following challenges to maintain the flexibility, scalability and performance of the data plane.

Hiding Heterogeneity: How to present a unified data plane abstraction and interfaces that hide the underlying differences between hardware and software switches to the controller?

Maintaining Performance: How to design and program the additional stateful processing modules to maintain the line rate processing and minimize latency overhead?

Data Plane Generality: How to build a general and reconfigurable hardware architecture that can accommodate many types of stateful network functions through simple reconfigurations from the controller?

The following sections will explain how we address the above challenges with our hardware design.

1) Unified Data Plane Abstraction and Interfaces: Different networks may choose to deploy software switches, hardware switches, or both types. To reduce the complexity of application deployment and management, we keep the three-table abstraction: State Table (ST), State Transition Table (STT), and Action Table (AT) on hardware switches similar to software switches, thus maintaining a consistent abstraction between hardware and software. Over the hardware switch, we implemented a software translation layer that is responsible for communicating with the controller just like a software switch. Therefore, the hardware and software switches can expose the same interfaces to the control plane.

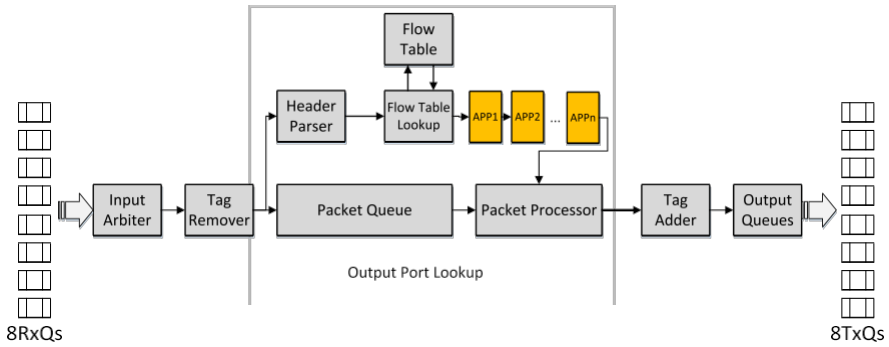


Fig. 7. Hardware packet processing pipeline in SDPA architecture.

2) Maintain High Throughput: One of the biggest challenges in the implementation of SDPA hardware switch is to maintain high throughput while ensuring correct processing logic. In one stateful APP module, all three tables including ST, STT, and AT should be looked up when processing one packet. One table lookup in TCAM requires 4 clock cycles (a design referenced from [5]), and three times of lookups would take 12 clock cycles. An APP module has to be locked for 12 clock cycles until the current package has been matched against all three tables, while the header parser module could deliver parsed header fields every 9 clock cycles. This inconsistency in clock cycle consumption could significantly decrease the throughput.

The key to address this challenge is to identify match dependencies between consecutive packets and shorten the clock cycle consumption inside each APP module. A match dependency occurs when an STT lookup triggers a state update into an ST entry, and the next packet belongs to the same flow and is being matched against that entry. In this case, the processing of the current packet must finish both STT lookup and ST update before the subsequent packet can be matched against ST. Above two steps must be strictly sequential to ensure logical correctness, while the rest stages create parallel opportunities to shorten clock cycle consumption. Therefore, we optimize the stateful App modules to allow ST lookup for the next packet immediately after the ST is updated by the current packet lookup in STT. And, the ST lookup for the next packet is parallel to the AT lookup for the current packet. In this way, the minimum gap between two consecutive packets is reduced to 9 clock cycles (4 cycles for ST lookup, 4 cycles for STT lookup and 1 cycle for ST update), which is the same as the header parsing stage. Therefore, SDPA architecture can realize line-rate processing for packets with any size (from 64 Bytes to 1500 bytes in our experiments).

3) Reconfigurable Hardware: Based on the unified SDPA paradigm and southbound APIs, SDPA hardware switches are more general than traditional dedicated middleboxes, and can support many stateful applications. For those applications that can be abstracted into the SDPA paradigm, SDPA hardware switches can support these applications by simply configuring ST, STT and AT. Through the unified interfaces, the controller is able to initiate a stateful application by issuing stateful table entries to the software layer on top of the hardware switch. The software layer will then translate them into hardware configurations and convey them into the hardware switch through drivers. To destroy a stateful application, the controller could clear all stateful table entries and steer packets away from the application, after which hardware resource can be reused to accommodate a new stateful application.

C. Customization of Network Function Chain

SDPA hardware switches support the customization of network function chains through the SDPA paradigm and corresponding APIs. In SDPA hardware switches, network functions can be deployed, updated and destroyed flexibly through configurations from the controller. For instance, to deploy a new network function to the SDPA hardware switch, we can initialize the application in the switches and configure the ST, STT, and AT through APIs from the controller. By accommodating several applications, we enable customization of network function chains within one switch and we can assign arbitrary processing orders of the applications as introduced above. We developed a sample network function chain in an SDPA hardware switch, which includes stateful firewall, DNS reflection defense, and heavy hitter detection.

In a DNS reflection attack, attackers send DNS requests to name servers using the victim host's source IP address, thus flooding the victim with the name servers' responses. To filter out these unsolicited responses, the SDPA switches of the victim network maintain 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 to 53 will be sent to the DNS reflection defense APP. In the ST, an unmatched DNS request will trigger the switch to install 2 new entries. We design four states in the state transition table: the initial state, request sent, response received for a previously recorded request, and the detection of an unsolicited response.

D. Dynamic Application Deployment on Hardware Switches

Applications that can be abstracted into the SDPA paradigm can be dynamically deployed on SDPA switches during runtime. If an application needs to maintain state information, and all of its actions belong to the SDPA-supported action set, it can be deployed on SDPA switches by sending configuration messages from the controller to the switches.

The new applications can be deployed in any position in the network function chain according to controller policies. It can be implemented through adjusting the parameter *n* of action GOTO_ST(*n*) in the flow tables to direct packets into the APP. It is deployed on SDPA hardware switches through the following steps. Firstly, according to application policy, the controller sends an encapsulated initialization message to SDPA switches which is used to install the state table *format*, the *entire* state transition table, and the action table *format*. Secondly, the software layer of the switch parses the messages issued by the controller and installs the tables into the hardware card. Thirdly, during runtime, if a new flow arrives at a switch, the switch will packet-in the first packet of the flow. The controller will add stateful table entries to the switches according to application decisions.

E. Scalability of SDPA Architecture

The ST, STT, and AT in SDPA could cost TCAM resources inside switches. Such an architecture could give rise to the scalability problems, including (1) Both ST and AT maintain the "Match Field", which could waste TCAM resources when processing massive flows; (2) Various applications would have different table length and depth. However, current FPGA implementation of SDPA has to assign those parameters in advance. This inflexibility would cause an inefficient resource allocation; and (3) Flow level states could be unavoidably massive. As a result, lots of table entries are required to maintain flow-level state inside the data plane and would heavily cost TCAM resources; (4) The STT of each stateful application should be pre-populated into the switches, which could cause major resource consumption due to massive states; Addressing above challenges, we propose the following solutions.

1) Duplicated Storage of "Match Fields": In SDPA design, each stateful application in the data plane processes packets according to related flow state. However, if an application wants to provide finer granularity of control over the flows, it may need to specify actions with regard to both packet match fields and its state. Suppose a stateful application wants to forward packets from subnet *A1* in *ESTABLISHED* state to *port 1*. It could install a rule on ST to monitor the packets from subnet *A1*, and install a rule on AT to forward the packets with the *ESTABLISHED* state to port 1. Thus, ST and AT both maintain the match fields of the same flow. Such an implementation on hardware could waste storage resources.

In fact, within an APP module, packet header fields are first matched against ST and then AT. We can utilize this knowledge by tagging flows in ST, an idea similar to the packet metadata that has been proposed in OpenFlow [20] and P4 [9], and identifying different flows in AT according to tags. More precisely, we extend ST with a field that records a tag for different flows in ST. Then, the combination of *state* and *tag* of a flow, which are precisely the fields to match in AT, are extracted and passed to AT to look up the actions. By tagging flows in ST, we can only store the match fields of the flows *once* in ST, in order to reduce the storage resource overhead.

2) Fixed Table Length and Depth: Currently, each application possesses three tables with fixed entry length and table depth in the hardware implementation due to the constraints of FPGA. However, according to our design, we can support any kind of stateful applications with various states. We cannot pre-estimate the header fields to match, the flow number to process and the entry number of the tables. Therefore, we have to allocate abundant resources just in case of heavy resource consumption of an application, which could cause waste due to extra entry length and table depth.

To address this challenge, we restrict the number of applications carried in each hardware switch to ensure sufficient resource for each application and avoid potential overload. In the future, we plan to implement SDPA on advanced hardware platforms such as RMT [10] that provides configurable width and depth of tables to improve resource utilization efficiency.

3) Massive Number of Flow-Level States: SDPA supports flow-level state monitoring in the data plane. However, flows could be of excessive number and result in heavy cost of data plane TCAM resources. This challenge seems inevitable due to our proposal of maintaining state inside the data plane. Nevertheless, according to a detailed research of commonly used stateful applications, we are able to recognize some particular states that are monitored by lots of applications. As can be seen in TABLE I, two most commonly monitored states are TCP state and flow counters. This reveals that we do not need to maintain separate table sets for different applications. Instead, we divide applications by the states they monitor, and keep one table set for applications monitoring the same state.

We develop a State Interest Registration (SIR) module inside the controller, which provides a set of state choices and collects state interests from all stateful applications. It will issue tables for each *type of state* into switches. All packet-ins will be sent to all applications to check what state of this flow they want to monitor and handle. Only *one ST entry* of this flow related to *one type of state* will be issued. State reports from switches are stored inside SIR. Through SIR, tables can be aggregated at application level, resulting in a major efficiency improvement on data plane resource utilization.

4) Pre-Population of STT: During the application initialization, the STT of an application should be pre-populated into the data plane, which could harm scalability. However, according to above summary, two most commonly monitored states are the TCP state and flow counters. We design 16 transition policies according to the TCP state machine and only 2 transition policies for flow counters introduced in Fig 4. Besides, if two applications follow the same state machine, the state transition policies of the two applications are likely to be identical, where only one STT is issued into the switch for the two applications to further save switch resources.

V. EVALUATIONS

We run the SDPA software switch in Ubuntu 12.04 system on a DELL R720 server equipped with a Xeon E5-2609 (2.4 GHz) CPU, 16GB internal memory and two 10 Gigabit Network Interface Cards. Furthermore, we use OVS-DPDK to enhance the performance of the software SDPA switch. We run an enhanced Floodlight controller on another server

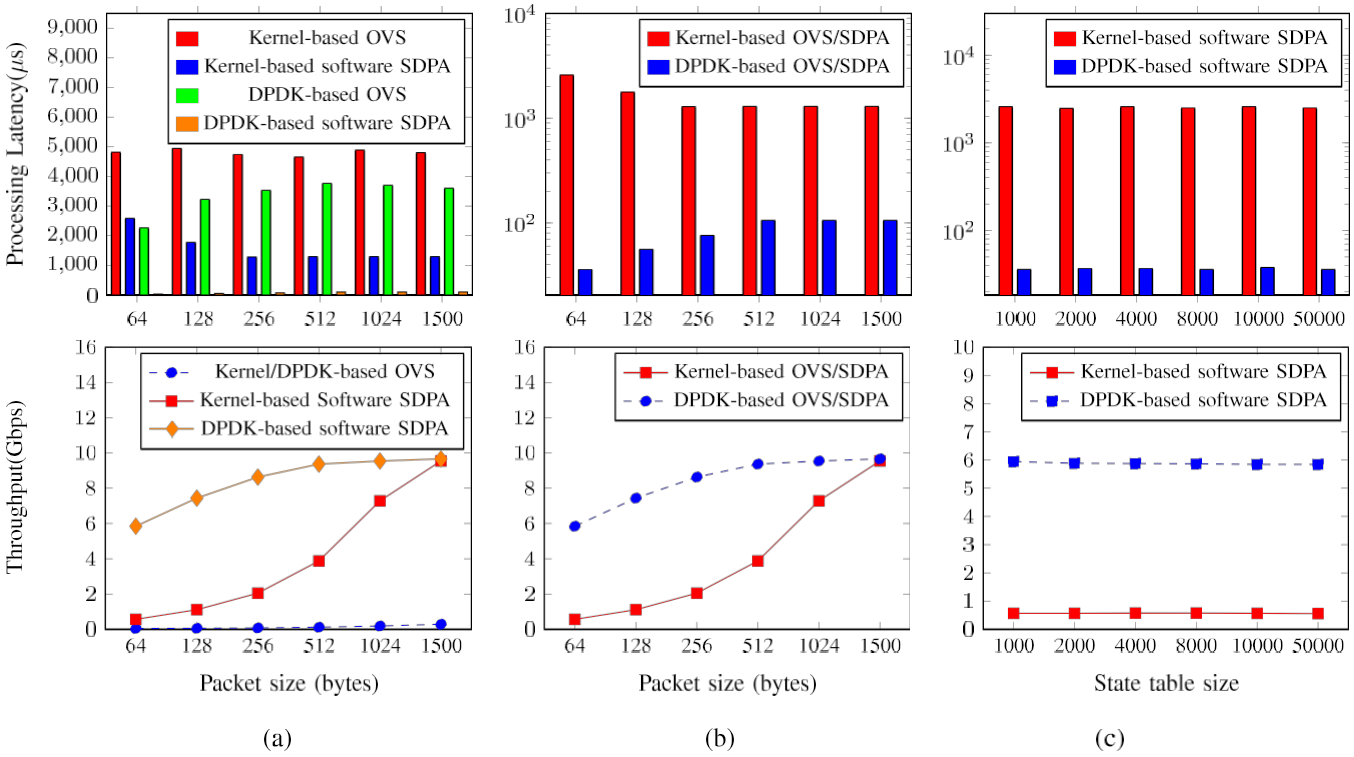


Fig. 8. Performance comparison between a SDPA software switch and a traditional OVS-based OpenFlow software switch with or without DPDK acceleration. Kernel-based stands for software implementation without DPDK acceleration. (a) Stateful firewall (SF) (b) Stateless forwarding (c) Scalability of software SDPA.

with the same configuration. The controller is wire-connected to software and hardware switches respectively in different experiments. SDPA hardware switch is implemented on ONetCard as introduced above. For test traffic, we use a DPDK based packet generator that runs on a separate server and is directly connected to the software or hardware switches. The generator sends and receives traffic to measure latency and throughput.

SDPA could efficiently support stateful applications. We evaluate SDPA with the following goals:

- demonstrate that the SDPA software implementation can support stateful applications with higher performance compared with OpenFlow software implementation (Fig. 8(a)), and suffers little performance overhead when supporting stateless applications (Fig. 8(b)). Besides, SDPA stateful processing only introduces a little performance overhead compared with stateless processing (Fig. 9(b)).
- demonstrate that SDPA has great scalability with respect to ST and AT sizes (Fig. 8(c)).
- compare the performance of SDPA hardware implementation and software implementation (Fig. 9(a)).
- demonstrate that SDPA software and hardware implementations could support service chains with linear latency and equal throughput (Fig. 9(c)).

A. Performance of Stateful Firewalls in SDPA Software Switch v.s. in Traditional Openflow Software Switch

To evaluate the efficiency of SDPA, we developed a stateful firewall application based on the traditional SDN architecture, where the state information is maintained in the controller. In this architecture, a large number of packets should be sent to the controller to check its state information before forwarding. We evaluated the performance of processing states in software switches in the SDPA architecture against processing states in the controller in the traditional SDN architecture.

We tested the forwarding latency and the throughput respectively by sending packets of 64 to 1500 bytes. The average forwarding latency reduces significantly in SDPA architecture than that in traditional SDN architecture, with or without DPDK acceleration, as shown in Fig. 8(a). In addition, the throughput increases significantly in the SDPA architecture. When realizing stateful firewalls in the traditional SDN architecture, the processing bottleneck of the controller limits the processing capability of the firewalls. While realizing stateful firewalls in the SDPA architecture, SDN data plane maintains all state information. The throughput of the firewalls is significantly improved regardless of the size of packets.

B. Performance of Stateless Forwarding in SDPA Software Switch v.s. in Traditional Openflow Software Switch

Since the SDPA architecture is fully compatible with OpenFlow, SDPA can also support *stateless* processing. While performing stateless forwarding in the data plane, the average forwarding latency in the SDPA architecture remains almost equal to that in the traditional SDN architecture with or without DPDK acceleration, as shown in Fig. 8(b). The throughput in the SDPA architecture is almost the same as that in the traditional SDN architecture. Applications that do not need to maintain state information can be fully supported by simply

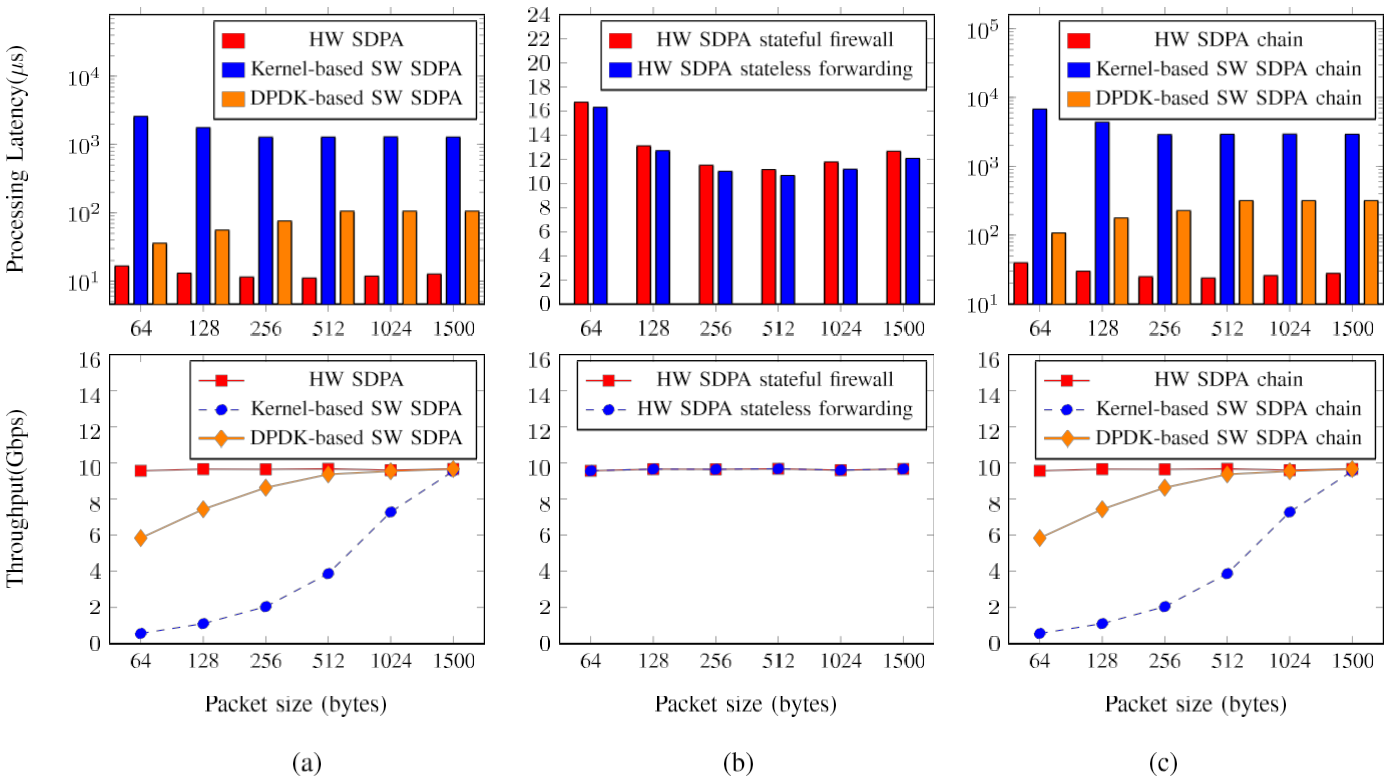


Fig. 9. Performance comparison between a SDPA hardware switch and a SDPA software switch. In the figure legends, HW is short for Hardware, SW for Software, Kernel-based for software implementation without DPDK acceleration. (a) Stateful firewall (b) Stateful firewall v.s. Stateless forwarding (c) Network function chain.

sending packets through flow tables like traditional OpenFlow with no additional overhead.

C. Testing the Scalability of State Tables

We performed a test on the scalability with respect to the state table size in the SDPA software switch and its impact on the latency and throughput. Since state tables are implemented in SRAM in software prototype, the state tables size can be increased to a large extent. We used 64-byte packets to conduct our experiment. As the state table size increases from 1000 to 50000, the forwarding latency does not increase significantly, and the throughout almost remains the same with or without DPDK acceleration as shown in Fig. 8(c). The table lookup procedure consumes short time and has little effect on the performance, showing good scalability.

The hardware implementation of SDPA is developed based on ONetCard, where state tables are implemented in TCAM. For each flow, it takes 248 bits to store its header, 8 bits to store its state in the ST, and 320 bits to store an entry in AT. STT is small (24 bits for each entry) with the fixed entry number and is shared among all flows. Its average resource cost for one flow can be omitted. Therefore, the total TCAM consumption for one flow is 576 bits. Considering that OpenFlow 1.0 uses 568 bits for each flow table entry, memory consumption to maintain and process the state of one flow in SDPA is almost the same as a traditional flow entry. In the situation where one switch is fully assigned to carry a stateful firewall application, only one wildcard flow table entry that directs all packets into ST is needed. The state table can be used to perform state validation and forwarding, covering the role of the flow tables. In this way, little additional memory is needed to process the same number of flows compared with traditional OpenFlow.

D. Performance of Stateful Firewalls in the SDPA Hardware Switch v.s. in the SDPA Software Switch

We compared the performance of the stateful firewall application in SDPA hardware and software switches. As illustrated in Fig. 9(a), the forwarding latency of the hardware-based implementation is much lower than that of the kernel-based software implementation, while the throughput of the hardware-based implementation increases to a large extent compared with the kernel-based implementation, especially for small sized packets. Even with DPDK acceleration, software SDPA switches still underperforms hardware switches in latency and throughput. The experimental results demonstrate the feasibility of implementing SDPA paradigm in a hardware paradigm to achieve much higher performance than software implementation. The SDPA hardware switch can achieve *line-rate* for packets of any size during stateful packet processing.

E. Performance of Stateful Firewalls v.s. Stateless Forwarding in SDPA Hardware Switch

We compared our stateful firewall in the SDPA hardware switch with *stateless* forwarding in the traditional OpenFlow hardware switch. As can be seen in Fig. 9(b), the forwarding latency of the stateful firewall in the SDPA architecture is slightly higher than stateless forwarding. The processing

overhead is acceptable and the throughput is nearly unchanged as shown in Fig. 9(b). This proves that stateful processing contributes to only a little performance overhead. As explained above, it only takes as few as 9 additional clock cycles to perform table lookup and processing in the data plane, which contributes to only 72*ns* for 125MHz board clock frequency.

F. Performance of the Network Function Chain in SDPA Hardware Switch v.s. in SDPA Software Switch

We evaluated the performance of the network function chain comprising stateful firewall, DNS reflection defense, and heavy hitter detection functions implemented on the SDPA hardware switch against the same chain on software SDPA switch. As can be seen from Fig. 9(c), average forwarding latency of hardware switch is much lower than that of software switch, while the throughput of hardware is much higher than software. This result is consistent with Section V-D, and proves the feasibility of forming a hardware SDPA network function chain and achieving high performance.

Due to the limited hardware resource, we could only implement three NFs on the ONetCard platform. By comparing Fig. 9(a) and Fig. 9(c), we could conclude that increasing the number of Apps traversed by packets could increase the processing latency and incur little overhead on the throughput for both hardware and software implementations. As analyzed in Section V-C, the table entry size in SDPA is fixed. Therefore, the number of Apps that can be accommodated on a hardware platform is proportional to the hardware resource amount. In future, we plan to implement SDPA on advanced hardware platforms that could provide more richer resource.

G. Evaluation Based on a Real-World Network Topology

We evaluated the performance of software network function chain in a Mininet simulation environment based on a real-world network topology derived from the Stanford backbone network [2]. We selected a three-hop forwarding path. Each switch in this path carries one unique network function. For packets of 1024 bytes, the forwarding latency is approximately 3900μs and the throughput is about 9.5Gbps. This result demonstrates the feasibility of implementing SDPA in a real-world network topology with satisfying performance.

VI. Related Work

OpenFlow Data Plane Abstraction: Some research efforts have been recently devoted to extend the OpenFlow data plane abstraction [8], [10], [13], [19], [29]. Bosshart *et al.* [10] pointed out that the rigid table structure of current hardware switches limits the scalability of OpenFlow packet processing to 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. In comparison, we strive to enhance the programmability of the data plane by adding a co-processing unit in SDN switches.

Bianchi *et al.* [8] proposed an abstraction to formally describe the desired stateful processing of flows inside SDN data plane based on eXtended Finite State Machines (XFSM). The functionality of XFSM table in OpenState is similar to that of STT and AT tables in SDPA. The design of XFSM actually combines the STT and AT of SDPA. However, OpenState only supports *finite* state machines, while SDPA could also accommodate *infinite* state machines by enhancing definitions of the *event* and *action* fields in STT and AT tables. Therefore, SDPA provides a more powerful and general abstraction for various stateful applications.

Moshref *et al.* [21] proposed FAST (Flow-level State Transitions) as a new switch primitive for SDN. Shuyong *et al.* [29] introduced a preliminary stateful forwarding solution in the SDN data plane. However, none of them presented the relationships and interactions between the state tables and flow tables in SDN switches. Thus their compatibility with OpenFlow remains unclear. They also did not elaborate the fundamental shortcoming of the limited "match-action" paradigm in the 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 to operate the state information in the data plane. We also developed both software and hardware prototypes of the SDPA architecture. Especially, we developed three stateful applications and organized them as a network function chain in an SDPA hardware switch, and provided support for dynamic deployment of new applications.

Data Plane Programmability: Some efforts have been devoted to enhance the programmability of the SDN data plane [9], [27]. Among them, P4 [9] is a typical language for programming protocol-independent packet processors. Although P4 is also capable of supporting advanced applications, such as the heavy hitter detection and the Paxos consensus protocol [11], in the data plane, SDPA and P4 differ in their design goals. P4 aims to enhance the data plane programmability. Therefore, it proposes an abstraction for *data plane behaviors* along with a related high level language. In contrast, SDPA addresses the limitation of the simple "match-action" paradigm of OpenFlow when supporting advanced stateful applications. Since heavily relying on SDN controllers for all state maintenance incurs both scalability and performance issues, SDPA proposes a "match-state-action" paradigm for *stateful applications*, which could be intuitively programmed and efficiently supported in the data plane. Thus, SDPA provides a higher level abstraction than P4. Through proper encapsulation, P4 could work as a potential target data plane for SDPA. Moreover, P4 data plane supports customized protocols by parsing headers based on a state machine. Since SDPA data plane already supports state machines, SDPA could possibly achieve the protocol independence similar to P4.

Middlebox Enhancement: Since current OpenFlow data plane is limited to support stateful processing, the advanced packet processing has been turned to specialized middleboxes [7], [16]. Anwer et al [7] 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, which attempts to combine

traditional middleboxes with the SDN architecture. There are also some efforts for developing middlebox functions using SDN [14], [22], [26]. 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.* [22] proposed to add an SDN-based policy enforcement layer for efficient middlebox-specific traffic steering. However, above research efforts lack a general programming interface for stateful applications. Moreover, the network is filled with various middleboxes, and the network structure 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 networking functions via software, as opposed to having to rely on proprietary middleboxes to handle these functions, referred to as Network Functions Virtualization (NFV) [17]. Since SDN and NFV are complementary technologies [30], 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 [6]. Especially our hardware implementation can provide high forwarding capacity to fulfill the requirements of stateful packet processing required by advanced network functions.

VII. Discussion

A. Flow Migration in SDPA

In the situation where one switch is overloaded, the operator needs to migrate some flows from the switch to another one. For stateful applications such as stateful firewalls, migrating flows means migrating the ST, STT and AT entries. We could refer to OpenNF [15] and enhance both SDPA data plane and control plane to realize loss-free and order-preserving state migration among SDPA switches. As introduced above, the software layer of a SDPA hardware switch is responsible for the communication with the controller. Therefore, software and hardware switches will react similarly in state migration.

B. Limitation of SDPA

As discussed above, SDPA could efficiently support applications that can be abstracted into finite or infinite state machines. Thus, the capability of SDPA equals a Finite Automation in the automation theory, and therefore is not Turning Complete. However, the stateful abstraction of SDPA is capable of representing a large number of general network functions including stateful firewalls, heavy hitter detection, etc. Through both software and hardware implementations, SDPA could support them with high performance.

VIII. Conclusion and Future Work

In this paper, we have 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. We have proposed a novel stateful data plane architecture SDPA. Through adding a co-processing unit, the 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. In addition, we have implemented both software and hardware prototypes of SDPA switches, and developed a network function chain in a SDPA hardware switch. Experimental results show that the SDPA architecture can tremendously improve the forwarding efficiency with manageable processing overhead for those applications that need to maintain states.

For the future work, we will develop more stateful applications based on the SDPA architecture to further validate the effectiveness of our approach. We will also extend the concept of state in our architecture to include switch states such as queueing delay, link states, and other customized states to support more complex applications. Finally, future adoption of advanced data plane platforms, such as RMT [10], could further improve data plane resource utilization efficiency and enhance the scalability of SDPA architecture.

IX. Availability

The SDPA source code is available on Github at: https://github.com/sdpa-project/sdpa.git

References

[1] *Floodlight*, accessed on Jul. 22, 2017. [Online]. Available: http://www.projectfloodlight.org/floodlight/

[2] *Header Space Library*, accessed on Jul. 22, 2017. [Online]. Available: https://bitbucket.org/peymank/hassel-public

[3] *Onetcard*, accessed on Jul. 22, 2017. [Online]. Available: http://www.xilinx.com/products/boards-and-kits/1-411ymv.html

[4] *Open vSwitch*, accessed on Jul. 22, 2017. [Online]. Available: http://openvswitch.org/

[5] *OpenFlow Switch on NetFPGA*, accessed on Jul. 22, 2017. [Online]. Available: https://github.com/NetFPGA/netfpga

[6] *Service Function Chaining (SFC)*, accessed on Jul. 22, 2017. [Online]. Available: https://datatracker.ietf.org/wg/sfc/

[7] B. Anwer, T. Benson, N. Feamster, D. Levin, and J. Rexford, "A slick control plane for network middleboxes," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2013, pp. 147–148.

[8] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: Programming platform-independent stateful OpenFlow applications inside the switch," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 44–51, 2014.

[9] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

[10] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *Proc. ACM SIGCOMM Conf. SIGCOMM (SIGCOMM)*, 2013, pp. 99–110.

[11] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos made switch-y," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 46, no. 1, pp. 18–24, 2016.

[12] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using FlowTags," in *Proc. USENIX Symp. Netw. Syst. Des. Implement. (NSDI)*, 2014, pp. 533–546.

[13] Open Networking Foundation (ONF), "Software-defined networking: The new norm for networks," Open Netw. Found. (ONF), White Paper, 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf

[14] A. Gember, R. Grandl, J. Khalid, and A. Akella, "Design and implementation of a framework for software-defined middlebox networking," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 467–468, 2013.

[15] A. Gember-Jacobson *et al.*, "OpenNF: Enabling innovation in network function control," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 163–174.

[16] G. Gibb, H. Zeng, and N. McKeown, "Initial thoughts on custom network processing via waypoint services," in *Proc. 3rd Workshop Infrastruct. Softw./Hardw. Co-Des. (WISH)*, 2011. [Online]. Available: http://yuba.stanford.edu/~nickm/papers/waypoint-cgo11.pdf

[17] R. Guerzoni "Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action, introductory white paper," in *Proc. SDN OpenFlow World Congr.*, 2012, pp. 5–7.

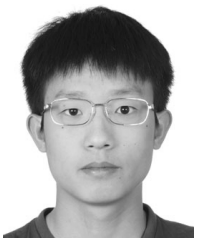
[18] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, and S. Goll, "Modeling and performance evaluation of an OpenFlow architecture," in *Proc. 23rd Int. Teletraffic Congr.*, 2011, pp. 1–7.
[19] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières, "Millions of little minions: Using packets for low latency network programming and visibility," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 3–14.
[20] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.
[21] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, "Flow-level state transition as a new switch primitive for sdn," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2014, pp. 61–66.
[22] Z. A. Qazi *et al.*, "SIMPLE-fying middlebox policy enforcement using SDN," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 27–38, 2013.
[23] C. Roeckl, "Stateful inspection firewalls," Juniper Netw., Sunnyvale, CA, USA, White Paper, 2004. [Online]. Available: http://www.abchost.cz/download/204-4/juniper-%EE%80%80stateful%EE%80%81-inspection-firewall.pdf
[24] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implement.*, 2012, p. 24.
[25] J. Sherry *et al.*, "Making middleboxes someone else's problem: Network processing as a cloud service," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 13–24, 2012.
[26] S. Shin *et al.*, "FRESCO: Modular composable security services for software-defined networks," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2013. [Online]. Available: https://www.internetsociety.org/doc/fresco-modular-composable-security-services-software-defined-networks
[27] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2013, pp. 127–132.
[28] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *IEEE Commun. Mag.*, vol. 51, no. 2, pp. 136–141, Feb. 2013.
[29] S. Zhu, J. Bi, and C. Sun, "SFA: Stateful forwarding abstraction in SDN data plane," Open Netw. Summit Res. Track (ONS), 2014.
[30] M. Zimmerman, D. Allan, M. Cohn, N. Damouny, and Kolias, "OpenFlow-enabled SDN and network functions virtualization," ONF, White Paper, 2014. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/solution-briefs/sb-sdn-nvf-solution.pdf

Chen Sun received the B.S. degree from the Department of Electronic Engineering, Tsinghua University, in 2014, where he is currently pursuing the Ph.D. degree with the Institute for Network Sciences and Cyberspace. He has authored papers in SIGCOMM, ICNP, SOSR, *IEEE Communications Magazine*, and the *IEEE Network Magazine*. His current research interests include Internet architecture, software-defined networking, and network function virtualization.

Jun Bi (S'98–A'99–M'00–SM'14) received B.S., C.S., and Ph.D. degrees from the Department of Computer Science, Tsinghua University, Beijing, China. He is currently a Changjiang Scholar Distinguished Professor and the Director of Network Architecture Research Division, Institute for Network Sciences and Cyberspace, Tsinghua University. His current research interests include Internet architecture, SDN/NFV, and network security. He successfully led tens of research projects, published over 200 research papers and 20 Internet RFCs or drafts, and also holds 30 innovation patents. He received the National Science and Technology Advancement Prizes, the IEEE ICCCN Outstanding Leadership Award, and Best Paper awards. He is the Co-Chair of the AsiaFI Steering Group and the Chair of the China SDN Experts Committee. He served as the TPC Co-Chair of a number of Future Internet related conferences or workshops/tracks at INFOCOM and ICNP. He served on the Organization Committee or Technical Program Committees of SIGCOMM, and ICNP, INFOCOM, CoNEXT, and SOSR. He is a Distinguished Member of the China Computer Federation.

Haoxian Chen received the B.S. degree from the Department of Electronic Engineering, Tsinghua University, Beijing, China, in 2016. He is currently pursuing the Ph.D. degree with the Computer Science Department, Carnegie Mellon University. His current research interests include computer networks and distributed systems.

Hongxin Hu (S'10–M'12) received the Ph.D. degree in computer science from Arizona State University, Tempe, AZ, in 2012. He is currently an Assistant Professor with the Division of Computer Science, School of Computing, Clemson University. His current research interests include security in emerging networking technologies, security in Internet of Things, security and privacy in social networks, and security in cloud and mobile computing. He has authored over 80-refereed technical papers, many of which appeared in top conferences and journals. He was a recipient of the Best Paper Award from ACM CODASPY 2014, and the Best Paper Award Honorable Mentions from ACM SACMAT 2016, IEEE ICNP 2015, and ACM SACMAT 2011. His research has been funded by the National Science Foundation, U.S. Department of Transportation, VMware, Amazon, and Dell. He has served as a Technical Program Committee Member for many conferences, such as the IEEE Conference on Communications and Network Security, the ACM Symposium on Access Control Models and Technologies, and the IEEE Global Communications Conference.



Zhilong Zheng received the B.S. degrees from the School of Software Engineering from Chongqing University, Chongqing, China, in 2016. He is currently pursuing the Ph.D. degree with the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include software-defined networking and network function virtualization.

Shuyong Zhu received the B.S. and M.S. degrees from the National University of Defense Technology, Changsha, China, and the Ph.D. degree with from Department of Computer Science, Tsinghua University, Beijing, China. His research fields include Internet architecture, software-defined networking, and network function virtualization.

Chenghui Wu received the B.S. degree from the Department of Electronic Engineering, Tsinghua University, in 2013, where he is currently pursuing the Ph.D. degree with the Institute for Network Sciences and Cyberspace. His research interests include Internet architecture and software-defined networking.