UniSec: a Unified Security Framework with SmartNIC Acceleration in Public Cloud*

Jinli Yan Lu Tang Junnan Li Xiangrui Yang Wei Quan Hongyi Chen Zhigang Sun
College of Computer, National University of Defense Technology
{yanjinli10,lutang,lijunnan,yangxiangrui11,w.quan,chenhongyi,sunzhigang}@nudt.edu.cn

ABSTRACT

In the public cloud, the software security functions that multi-tenants deploy in their virtual networks have limited performance. SmartNIC overcomes these limitations by implementing these security functions with hardware acceleration. However, the shared SmartNIC resources are not open for external users with security considerations. Since the security requirements of tenants are diverse, it is tedious for network operators to develop these functions from scratch with low-level APIs.

This paper presents UniSec, a unified programming framework for fast security functions development while improving performance with SmartNIC acceleration. UniSec provides modular abstraction for a single function and module sharing among multiple security functions. With the well-defined APIs of UniSec, developers only need to focus on the core logic instead of complex underlying operations including resource management, matching algorithms, etc. Experimental results show that the code has been reduced by 65% on average for each security function with UniSec. UniSec also improves processing performance up to 76%, compared with the software-only implementation.

KEYWORDS

SmartNIC, Security Function, Programming Framework

ACM Reference Format:
Jinli Yan Lu Tang Junnan Li Xiangrui Yang Wei Quan Hongyi Chen Zhigang Sun. 2019. UniSec: a Unified Security Framework with SmartNIC Acceleration in Public Cloud. In *ACM Turing Celebration Conference - China (ACM TURC 2019) (ACM TURC 2019), May 17–19, 2019, Chengdu, China.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3321408.3323087

*Lu Tang is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACM TURC 2019, May 17–19, 2019, Chengdu, China
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7158-2/19/05…$15.00
https://doi.org/10.1145/3321408.3323087

1 INTRODUCTION

The public cloud provides tenants with on-demand provisioning of computing and storage resources, which reduces considerable costs in capital expense and operating expense [4]. Thus many tenants migrate networks into the public cloud, and it is critical to protect their network from attack [18]. A traditional solution is deploying the proprietary security middleboxes alongside switches. However, the rigid middleboxes without high programmability cannot meet various security functions requirements of multi-tenants [1].

With the application of Network Function Virtualization [8], the flexible software-based security functions on off-the-shelf hardware are deployed by cloud providers (*i.e.*, Amazon, Microsoft, VMWare) to meet on-demand scaling and provisioning. However, existing software functions require multi-core to achieve 10GE rate [12]. Although Data Plane Development Kit [10] and Vector Packet Processor (VPP) [13] have successfully provided 40GE/100GE line speed in L2/L3 forwarding, the complex software security functions (*i.e.*, stateful firewall and Snort [3]) still have limited throughput and induce high latency.

To improve processing performance in software and retain flexibility, accelerating software functions with Graphics Processing Units (GPU) [9], Field Programmable Gate Arrays (FPGA) [12], Network Processor (NP) [12] becomes popular recently. Compared to other programmable hardware, FPGA uses non-von-neumann architecture and has lower processing latency. Moreover, FPGA is inexpensive and power-efficient. Recently, FPGA is designed as Smart Network Interface Card (SmartNIC) to accelerate security functions in Azure by offloading security functions[5, 6]. However, the SmartNIC resources are not open for external users with security considerations. Even if these resources are provided for them, it requires professional hardware/software co-design ability that most tenants do not have. As for the network operators, it is time-consuming and tedious to develop functions from scratch with low-level APIs for meeting diverse security requirements.

In this paper, we present **UniSec**, a **Uni**fied **Sec**urity framework for fast development and performance improvement

Table 1: Comparison between Existing Researches and UniSec Framework for Security Functions

Research	Software or Hardware	Function Orchestration	Development API	Deployment	SDN Architecture
OFX [17]	SW	✓	✓	Switch	✓
HEX [15]	HW(FPGA)	✓	✗	Switch	✓
OpenBox [2]	SW or HW Accelerator	✓	✓	Middlebox	✓
ClickNP [12]	SW and HW(FPGA)	✗	✗	Host	N/A
VFP [6]	SW and HW(FPGA)	✓	✗	Host	✓
UniSec	SW and HW(FPGA)	✓	✓	Host	✓

with SmartNIC acceleration. UniSec achieves this goal from three aspects. First, UniSec provides three modular abstraction models for single security function and general principles to map modules onto CPU and SmartNIC. Second, UniSec extracts modules shared among security functions to perform at the pre-process stage. Without executing repetitive logic, the processing path of multiple security functions is shortened. In addition, the flow-based security function orchestration is supported for meeting various deployment requirements. Third, UniSec provides a well-defined API to hide generic low-level logic into platform. The general logic includes resource management, match algorithms, etc. Thus the developers only need to focus on designing core processing logic and data structures. Moreover, the modules developed with the same API can be reused easily.

We implement UniSec prototype on CPU-SmartNIC platform (based on FAST framework [14]) and develop three security functions including Packet Filtering Firewall (PFW), Stateful Firewall (SFW) and Intrusion Detection System (IDS). Compared to the security functions developed from scratch without module reuse, UniSec reduces the code by 65% on average. Experimental results show that UniSec improves performance up to 76% when comparing with software-only implementation.

The rest of this paper is organized as follows. We introduce the background and requirements in §2. We present the design of UniSec in §3. §4 presents evaluation results, followed by the conclusion in §5.

2 BACKGROUND AND GOALS

2.1 Background

Security Function. The complex processing flow of security functions is abstracted into "rule management⟶packet processing⟶log management". We take Snort [3] as an example. In the first stage, security rules are parsed and configured. Each security rule in Snort includes a rule header and multiple rule bodies. The rule header and bodies describe flow and attack features respectively. In the packet processing stage, Snort usually performs exact matching and regex matching on a packet for multiple times. When an attack is detected, the alert messages are generated. In the last stage, these messages are collected and analyzed in local or remote servers. Thus many security functions are more complex in processing logic and rules when compared with network functions.

In real-world deployment, the policy of users usually involves multiple security functions. That means a packet needs to go through these security functions one by one (security function chain). Besides, the requirements for security functions of multi-tenants are various. Therefore, designing a programming framework should consider single function and security function chain.

Related Works. To the best of our knowledge, there are many existing works focusing on developing network functions and security functions. We compare some typical programming frameworks with UniSec in Table 1. Almost all the frameworks are designed based on SDN for high-efficiency function management. OpenBox [2], ClickNP [12], and VFP [6] do not make a distinction between network functions and security functions. Although OFX [17] provides flexible software API specific to security functions, it is costly to replace traditional ASIC switch with programmable switch. OpenBox [2] only provides a northbound API to assemble available processing blocks. HEX [15] proposed to implement the whole security processing into FPGA, which occupies high resource usage for complex security functions. ClickNP [12] and VFP [6] try to provide a framework with hardware/software co-design. ClickNP focuses on addressing the programming challenges with high-level language, but it does not support abundant primitives about state management [16]. Besides, ClickNP focuses on single function optimization. VFP requires network operators to offload actions of security functions into SmartNIC. However, it does not provide a well-defined API for third-party developers. UniSec aims to provide a programming framework on CPU-SmartNIC platform for network operators to provide high-performance security functions for tenants.

2.2 Design Goals

By analyzing various programming framework, we identify four key design goals for our framework.

G1:*Rational function division between CPU and SmartNIC.* Although the performance achieves maximum if all the operations are implemented in hardware, it is impractical with the limitations of developing efforts and resource usage in many cases. Thus it is important to modularize security functions with their features.

G2:*Flow-based dynamic security function orchestration.* The security detection requirements for different flows are various. To the best of our knowledge, almost all security functions only read packets without any modification. Thus the security function chain is only related to the flow feature, which makes it possible to implement orchestration before security processing.

G3:*Improving packet processing performance.* In addition to accelerate functions by offloading operations, we observe that many security functions have shared operations. We try to shorten the processing path by avoiding executing repetitive logic.

G4:*Providing well-defined programming API.* Apart from performance, tenants also care about development efficiency. Especially for complex security functions, improving the developing efficiency is more important because the operations that SmartNIC accelerates are limited. Therefore, a well-defined programming API should hide complex underlying operations in the platform.

3 UNISEC DESIGN

The framework of UniSec is depicted in Figure 1. UniSec uses a logically centralized controller to manage security functions for high efficiency. After receiving security rules from controller, the agent dispatches the rules to different security functions. We divide each security function into a virtual security function (vSF) and a hardware security function(hSF). In addition to packet processing, the vSF is used to configure hardware rules into hSF. We provide three modular abstraction models for single security function in Section 3.1. UniSec supports cooperation among multiple security functions on the same device, which reduces the latency brought by extra hops between devices. In order to reduce the number of I/O times, packets go through the pipeline of hSFs firstly, followed by the corresponding vSFs. The vSwitch is used to forward packets among vSFs. The optimization on security function chain is described in Section 3.2. Moreover, UniSec provides a unified API for the development of security function in Section 3.3.

3.1 Modular Abstraction of Single function

The entire processing flow of security function is described in Section 2.1, including "rule management →packet processing →log management". We simplify hardware-software co-design with modularization. The modules are classified into control modules and data modules. The control modules involve rule management and log management while data modules are in packet processing stage. We classify security functions into three types according to processing features and provide a modular abstraction model for each type in Figure 2.

Model 1: This model is designed for stateless header-based detection security function, e.g. PFW, DDoS Detection. The packet processing is abstracted into "Protocol parse→Header-based Match→Action". The protocol parse is to identify and extract appropriate fields used in subsequent stage. The header-based match is used to lookup in the rule table configured by rule management module. When a flow entry is hit, the corresponding action is executed and log infos are generated. Since security functions are mainly applied to filter anomaly traffic, the action mainly contains *pass* and *drop*. Finally, these infos will be collected and analyzed by log management module.

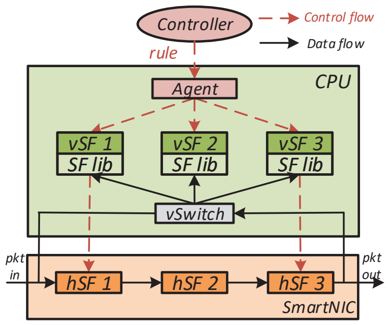
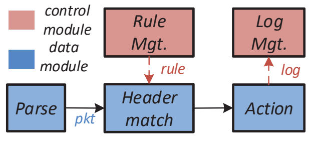


Figure 1: Overview of UniSec.

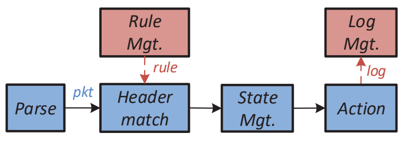
Model 2: This model is specific to stateful header-based detection security function, e.g. stateful firewall. Based on model 1, model 2 inserted a state management module between header-based match and action. This module is used to manage the historical state of flow (such as tcp flag, sequence number). If the states of current packet conflict with the previous state, the packet would be dropped.

Model 3: This model is relative to payload-based detection security function, e.g. IDS, WAF. The packet match is divided into header-based match and payload-based match. The header-based match is used to filter the eligible flows into payload-based match module. Thus the packets belonging to other flows will be transmitted to the next security function or physical port.

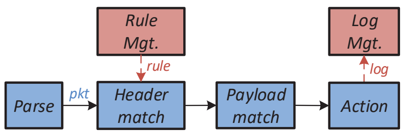
How to map these modules into CPU-SmartNIC has an important influence on the performance of security functions. These control modules are implemented in software. Regarding to data modules, the developers should consider performance, resource usage and developing efforts. Generally, the header-based operations are suitable to be implemented in SmartNIC while payload-based operations are in



(a) Stateless header-based detection



(b) Stateful header-based detection



(c) Payload-based detection

Figure 2: Three Modular Abstraction Models for Single Security Function.

CPU. The reason is payload-based processing involves complex matching algorithms and plentiful security rules when compared with header-based processing, which consumes lots of resources. This principle is provided for developers as a reference. In some cases, it is practical to offload part of payload-based operations into SmartNIC with sufficient resources. Besides, the state management operations may be moved to CPU if there are many types of states to store.

3.2 Module Sharing for Multi-Functions

In practical scenarios, different flows need to go through a security function chain. We try to shorten the processing path of security function chain while providing flow-based security function orchestration. Apart from protocol parse, we observe that L3-L4 header-based matching (such as five-tuple filtering) is also shared among security functions. We use two steps to avoid executing these operations repeatedly. First, we extract these operations as shared hSF in pre-process stage. The shared vSF is used to configure rules into shared hSF. Second, a global unique index **fid** is used to represent the packet feature. The **fid** is allocated by either controller or agent.

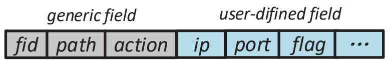


Figure 3: Metadata.

We attach a metadata in front of each packet to transmit the intermediate results. The metadata composes of generic fields and user-defined fields in Figure 3. The generic fields include **fid**, **path** and **action**. **fid** is used to distinguish different flows while **path** involves the information of security function chain. The **action** includes *To_CPU*, *To_PORT* and *DROP*. The user-defined fields are used to store the intermediate results. The information that former security functions extracted or generated can be reused in subsequent security functions.

We take an example to illustrate PFW+IDS chain in Figure 4. The user policy is that the http packets from A to B go through PFW(NO.1) and IDS(NO.3) successively. The agent computes the **fid**, **path** and rules with this policy. Then these rules are dispatched to Shared vSF, PFW vSF and IDS vSF respectively. The classification table in Shared vSF stores **key** (flow feature), **fid** and **path**. When the packets from A to

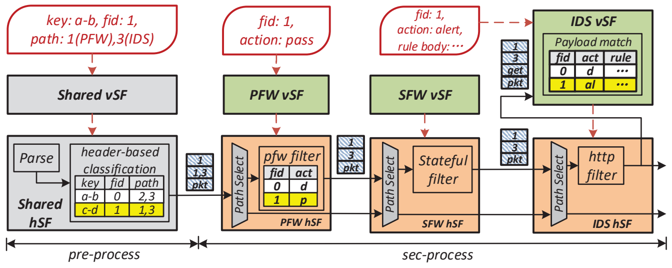
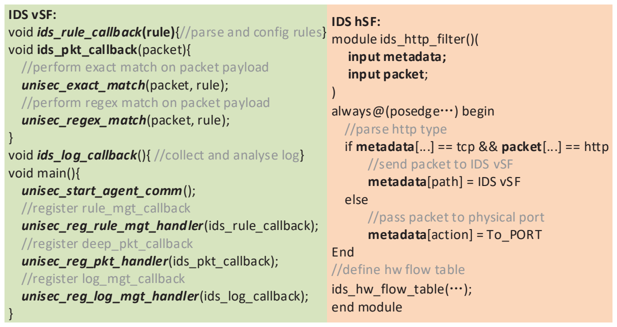


Figure 4: PFW+IDS Chain.

B comes, the second flow entry is hit and the **fid**, **path** [1] is set 1 and (1,3) respectively. The path selector module in each hSF uses **path** to decide whether this packet should be processed by this hSF. The PFW hSF performs *pass* action on this packet. Since the path does not involve SFW, the path selector in SFW hSF transmits this packet to IDS hSF directly. Then IDS hSF analyze whether it is a http packet and put the extracted http type (GET or POST) in the user-defined metadata for further payload-based detection in IDS vSF.



```
IDS vSF:
void ids_rule_callback(rule){//parse and config rules}
void ids_pkt_callback(packet){
   //perform exact match on packet payload
   unisec_exact_match(packet, rule);
   //perform regex match on packet payload
   unisec_regex_match(packet, rule);
}
void ids_log_callback(){ //collect and analyse log}
void main(){
   unisec_start_agent_comm();
   //register rule_mgt_callback
   unisec_reg_rule_mgt_handler(ids_rule_callback);
   //register deep_pkt_callback
   unisec_reg_pkt_handler(ids_pkt_callback);
   //register log_mgt_callback
   unisec_reg_log_mgt_handler(ids_log_callback);
}
```

```
IDS hSF:
module ids_http_filter()(
   input metadata;
   input packet;
)
always@(posedge···) begin
   //parse http type
   if metadata[...] == tcp && packet[...] == http
      //send packet to IDS vSF
      metadata[path] = IDS vSF
   else
      //pass packet to physical port
      metadata[action] = To_PORT
End
//define hw flow table
ids_hw_flow_table(···);
end module
```

Figure 5: Programming a Simplified Snort.

3.3 Modular Programming

We use modular design to develop security functions. The security functions are divided into platform-specific modules and user-specific modules. The platform-specific modules are used to hide complex low-level implementation and provide programming API for user-specific modules. We provide UniSec API to hide communication between security functions and agent, CPU resource management, complex matching algorithms and etc. Developers only need to register three user-defined callbacks to platform. The platform

[1] Here we use bitmap to represent the path information

would allocate CPU resource and create threads to run them. To be specific, *rule_mgt_callback*() is used to parse and config rules into rule tables of vSF and hSF. *deep_pkt_callback*() is used to perform complex logic (such as payload matching) with security rules. *log_mgt_callback* is used to collect and analyze log information. With respect to hSF, the packet parse and classification is platform-specific modules. Besides, we provide Stride BV matching algorthims [7] as IP blocks. The input of each hSF is metadata and packet. Therefore, users only need to design the hardware rule table and core state machine. We program a simplified Snort with UniSec in Figure 5.

4 IMPLEMENTATION AND EVALUATION

We implement UniSec on CPU-SmartNIC platform (based on FAST framework [14]). FAST provides a generic-purpose framework for hardware/software co-design. It hides the complex DMA, PCI-E and Linux kernel implementation for users. We extend Fast framework from three aspects to support our work. First, we implement protocol parse, packet classification and path selector module (about 2000 LoC in Verilog) in FPGA with Verilog and define **fid**, **path** and **action** fields in the metadata of FAST. Second, we provide UniSec Lib (about 500 LoC with C) by extending FAST API. Finally, we implement an agent (800 LoC with C) and security controller (2000 LoC with JAVA) respectively.

Table 2: Lines of Code(LoC) Comparison

Case	HW LoC		SW LoC		Reduction
	Strawman	UniSec	Strawman	UniSec	
PFW	2358	294	865	424	77.7%
SFW	5095	2129	921	465	56.9%
IDS	2873	809	2247	1201	60.7%

4.1 Development efficiency improvement

We implement three real-world security functions, including PFW, SFW and IDS. The implementation of them corresponds to three abstraction models in Section 3.1. PFW provides the five-tuple filtering. SFW provides the stateful filtering by analyzing TCP three-way handshaking and four-way close. We transplant payload-based detection logic of Snort to develop IDS. It supports exact matching and regex matching.

We use two ways to implement these security functions on CPU-SmartNIC platform. **Strawman:** We implement security functions based on FAST API without reusing any modules. **UniSec:** The security functions are developed based on UniSec framework without implementing shared modules (protocol parse, classification and etc.). We compare the lines of code (LoC) from software and hardware in Table 2. The results show UniSec achieves 65% reduction on average.

4.2 Performance Improvement

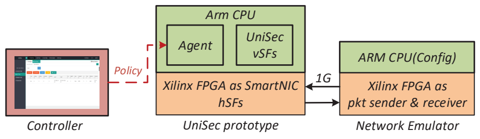


Figure 6: Experimental Setup.

We setup an experimental testbed to evaluate the performance improvement depicted in Figure 6. It includes a security controller, a UniSec prototype and a network emulator. The UniSec prototype and network emulator are built on a Xilinx Artix-7 FPGA connected to ARM Cortex A9 CPU (866MHz, Single core with two hardware threads) with PCI-E. In UniSec prototype, one thread is allocated to OS while the other is for running vSFs. All the links between UniSec prototype and emulator are 1GE fibers.

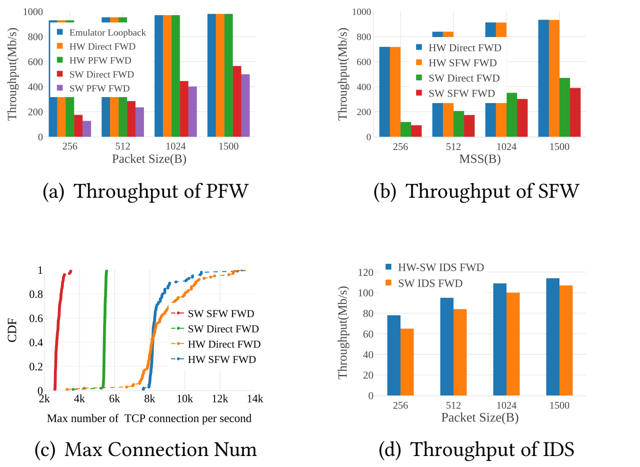


Figure 7: Performance of Single Security Function.

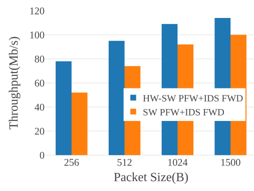
4.2.1 *Single Function Accleration.* First, we test the loopback throughput of network emulator (Emulator Loopback), the throughput of hardware direct forward (HW Direct FWD) and software direct forward (SW Direct FWD) as baseline in Figure 7(a). The throughput of Emulator Loopback and HW Direct FWD is line speed while that of SW Direct FWD is only 565 Mb/s at 1500B. The reason is that there are frequent buffer allocations/deallocations, memory copies between kernel and user space during SW Direct FWD. The overhead induced by these operations belongs to I/O overhead. The total overhead in software processing is composed of I/O overhead and packet processing overhead. In this paper, we only focus on reducing the packet processing overhead. we implemented the software version of PFW/SFW/IDS for comparison. Since the HW PFW FWD and HW SFW

FWD do not send packets to software, the performance improvement is computed by subtracting SW PFW/SFW FWD throughput from SW Direct FWD throughput.

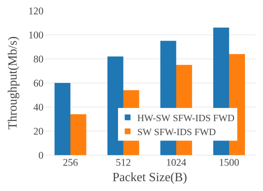
PFW. We test the throughput of HW PFW and SW PFW at 256/512/1024/1500B respectively in Figure 7(a). The performance improve is 13%-38%.

SFW. We use bandwidth estimation tool *Iperf* to start 8 TCP connections simultaneously. Then we test the throughput of hardware SFW forwarding (HW SFW FWD) and software SFW forwarding (SW SFW FWD) in different MSS (Max Segment Size) in Figure 7(b). HW SFW FWD improves 20.5%-28% processing performance when compared with SF SFW FWD. We use Apache HTTP server benchmarking tool *ab* to test max number of TCP connections per second and plot the CDF in Figure 7(c). The maximum value for HW SFW FWD and HW Direct FWD are more than 8000 while that for SW Direct FWD and SW SFW FWD are only 5500 and 2400 respectively when the probability is 80%.

IDS. We perform an exact matching and regex matching on the entire payload and adjust workload by changing packet size. The throughput of software IDS forward (SW IDS FWD) and hardware-software IDS forward (HW-SW IDS FWD) are in Figure 7(d). The performance difference between them decreases as the packet length increases. The reason is that the percentage of software processing overhead grows while that of hardware acceleration reduces. In most network scenarios, only specified flow are required to be detected by IDS. The performance efficiency also increases if many flows are forwarded in hardware directly.



(a) Throughput of PFW+IDS



(b) Throughput of SFW+IDS

Figure 8: Performance of Security Function Chain.

4.2.2 Security Function Chain Acceleration. We design two use cases about security function chain: PFW+IDS, SFW+IDS. The results of them are shown in Figure 8(a) and Figure 8(b) respectively. The performance improvement for PFW+IDS is 14%-50% while that for SFW+IDS is 26%-77%. The performance improvement of service chain is higher than that of single function because the repeated operations are left out in the processing path.

5 CONCLUSION

We present the design of a novel security programming framework with SmartNIC acceleration called UniSec. It simplifies the development of security function with user-friendly APIs while improving software processing performance. In the future, we will design a quantitative model for users to map functions into SmartNIC-CPU rationally. We believe UniSec will serve as a useful platform for network operators to customize security services for external tenants.

ACKNOWLEDGMENT

This research is supported by National Natural Science Foundation of China (Grant No.61702538, 61802417) and Research Project of National University of Defense Technology (Grant No.ZK17-03-53).

REFERENCES

[1] Benson et al. 2011. CloudNaaS: a cloud networking platform for enterprise applications. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 8.

[2] Anat Bremler-Barr et al. 2016. OpenBox: a software-defined framework for developing, deploying, and managing network functions. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 511–524.

[3] Cisco. 2019. Snort: Network Intrusion Detection and Prevention System. https://www.snort.org/.

[4] Paolo Costa et al. 2012. NaaS: Network-as-a-Service in the Cloud. In *Presented as part of the 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*.

[5] Firestone et al. 2018. Azure accelerated networking: SmartNICs in the public cloud. In *Proceedings of the 2018 USENIX NSDI*. 51–66.

[6] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *Proceedings of the 2017 USENIX NSDI*. 315–328.

[7] Thilan Ganegedara and Viktor K Prasanna. 2012. StrideBV: Single chip 400G+ packet classification. In *2012 IEEE 13th International Conference on High Performance Switching and Routing*. IEEE, 1–6.

[8] Bo Han et al. 2015. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine* 53, 2 (2015), 90–97.

[9] Sangjin Han et al. 2011. PacketShader: a GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 195–206.

[10] Intel. 2019. DPDK: Data Plane Development Kit. http://dpdk.org/.

[11] Muhammad Asim Jamshed et al. 2017. mOS: A reusable networking stack for flow monitoring middleboxes. In *Proceedings of the 2017 USENIX NSDI Conference*. 113–129.

[12] Bojie Li et al. 2016. ClickNP: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 1–14.

[13] Leonardo Linguaglossa et al. 2018. High-speed Software Data Plane via Vectorized Packet Processing. (2018).

[14] NUDT. 2019. Fast Project. http://www.fastswitch.org/.

[15] Park et al. 2018. HEX Switch: Hardware-assisted security extensions of OpenFlow. In *Proceedings of the 2018 Workshop on Security in Softwarized Networks: Prospects and Challenges*. ACM, 33–39.

[16] Pontarelli et al. 2019. FlowBlaze: Stateful Packet Processing in Hardware. In *Proceedings of the 2019 USENIX NSDI Conference*.

[17] Sonchack et al. 2016. Enabling Practical Software-defined Networking Security Applications with OFX. In *Proceedings of the 2016 NDSS*. 1–15.

[18] Vijay Varadharajan and Udaya Tupakula. 2014. Security as a service model for cloud environment. *IEEE Transactions on network and Service management* 11, 1 (2014), 60–75.