Computers & Security 124 (2023) 102976

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Extended data plane architecture for in-network security services in software-defined networks

Jinwoo Kim[a], Yeonkeun Kim[b], Vinod Yegneswaran[d], Phillip Porras[d], Seungwon Shin[c], Taejune Park[e,*]

[a] *Kwangwoon University, Republic of Korea*
[b] *S2W Inc., Republic of Korea*
[c] *KAIST, Republic of Korea*
[d] *SRI International, USA*
[e] *Chonnam National University, Republic of Korea*

A R T I C L E I N F O

Article history:
Received 19 July 2022
Revised 10 October 2022
Accepted 19 October 2022
Available online 25 October 2022

Keywords:
Software-defined networking (SDN)
Network security
Openflow
Data plane
Network function virtualization (NFV)

A B S T R A C T

Software-Defined Networking (SDN)-based Network Function Virtualization (NFV) technologies improve the dependability and resilience of networks by enabling administrators to spawn and scale-up traffic management and security services in response to dynamic network conditions. However, in practice, they often suffer from poor performance and require complex configurations because network packets must be 'detoured' to each virtualized security service, which expends bandwidth and increases network propagation delay. To address these challenges, we propose a new SDN-based data plane architecture, called DPX (Data Plane eXtension), that natively supports in-network security services. The DPX action model reduces redundant processing caused by frequent packet parsing and provides administrators with a simplified (and less error-prone) method for configuring security services into the network. DPX also increases the efficiency of enforcing complex security policies by introducing a novel technique called *action clustering*, which aggregates security actions from multiple flows into a small number of synthetic rules. Also, the application of action clustering (i.e., advanced and global) provides more diverse policies and network-wide detection. We present an implementation of DPX in hardware using NetFPGA-SUME and in software using Open vSwitch. We evaluate the performance of the DPX prototype and the efficacy of its flow-table simplifications against a range of complex network policies exposed to line rates of 10 Gbps.

1. Introduction

Today's dynamic network environments represented by 5G or cloud computing enable many services to be operated through networking, shifting the paradigm of service infrastructure. In order to efficiently manage various services, virtualization techniques are widely employed in the infrastructure to separate services from specialized hardware devices. The separation leads to the deployment of virtualized machines (VMs) that run on commercial off-the-shelf (COTS) servers, facilitating elastic scaling and dynamic resource provisioning. Because VMs can be up, down, or moved anytime, anywhere for the resource provisioning, security polices should be dynamically updated to the changes. However, the traditional networking architecture that is almost static has difficulty in adapting to the frequent changes due to its low-flexibility. More specifically, adjusting configurations of legacy network devices mainly relies on operator's manual labor, which is time-consuming and error-prone (Casado et al., 2007; Greenberg et al., 2005).

Therefore, new network architectures for adapting to network changes have emerged, such as Software-Defined Networking (SDN) and Network Function Virtualization (NFV). Both technologies aim to decouple network functions from hardware devices into the software so that operators can devise network polices flexibly. By doing so, SDN provides a unified platform that controls network devices in a centralized place, and NFV enables diverse network functions to be deployed to COTS servers. Thus, even if a network environment changes (e.g., VMs are created or migrated on the fly), operators can easily alter network configurations with SDN, or quickly deploy required network functions with NFV. Considering these benefits, many SDN/NFV-based security solutions have been proposed so far to defend existing

* Corresponding author.
E-mail address: taejune.park@jnu.ac.kr (T. Park).

https://doi.org/10.1016/j.cose.2022.102976
0167-4048/© 2022 Elsevier Ltd. All rights reserved.

network threats (e.g., DDoS attacks and network scanning) in a cost-efficient way (Fayaz et al., 2015; Fayazbakhsh et al., 2014; Gember-Jacobson et al., 2014; Hwang et al., 2014; Qazi et al., 2013; Shin and Gu, 2012; Shin et al., 2013).

However, deploying SDN/NFV-based solutions in the dynamic network environment raises performance and management issues. First, in the current deployment strategy, all traffic must be detoured to network *choke-points* where an SDN controller or NFV instance is deployed to enforce security policies. However, this comes at the cost of increasing routing hops, leading to significant performance degradation. Second, in order to fully leverage the benefits of SDN/NFV-based solutions, network operators need to carefully design an orchestration strategy on SDN control plane (Fayaz et al., 2015; Kim and Feamster, 2013). For example, network operators need to solve an optimization problem to determine optimal locations of NFV nodes given resources, or produce diverse network flow rules to steer traffic through the choke-points.

We posit that the root cause stems from the need for traffic steering when operating SDN/NFV security solutions. Thus, if we do not need to deploy additional nodes for running security functions, we can achieve the better performance and management simplicity. To that end, we raise the following research question: *Can we extend SDN switches to support native security functions from the data plane to eliminate the need for traffic steering?* To answer the question, we begin by examining the existing SDN data plane architecture. Currently, most SDN switch implementations merely support basic packet-handling logic (e.g., forward, drop, and modify headers). However, most SDN switches include various processing elements (e.g., storing packets and parsing headers) that may be utilized for embedding additional security functions. For example, if a switch filters out disallowed packets, the packets would avoid a trip to the firewall, which eliminates redundant packet forwarding. Recent advances in high-performance (i.e., high throughput, low latency) software switches Honda et al. (2015); Open vSwitch (2022), suggests that this could potentially be a feasible solution. Based on these insights, we design DPX (**D**ata-**P**lane E**x**tension), a new SDN data plane architecture that extends OpenFlow (2022), a de-facto standard protocol used for communication between an SDN controller and switches. By doing so, DPX not only allows network operators to easily enforce security policies (by only focusing the switch rules), but also achieves high-performance and low-latencies.

An important challenge to be addressed in making this leap is the ability to express security policies over aggregated flow sets because representing security rules using per-flow rules is prohibitively expensive (i.e., *flow-steering complexity challenge*). To address this problem, DPX also implements a novel technique, called *action clustering*, which allows a security service to concurrently operate on a set of flows. It not only simplifies a flow table for a service chaining, but also enables an advanced action clustering to represent more sophisticated policies, or a global action clustering to detect and mitigate a network-wide attack. In addition, we aim to keep the original philosophy of SDN (i.e., simple data plane), and thus we basically make DPX as modular components for an SDN data plane. As we noted, newly added security actions will be realized by OpenFlow actions, and those new security actions will be supported by each DPX security action block. Each action block can be easily inserted or removed based on requirements. For example, if a network administrator wants to detect DDoS attacks, we can provide DPX SDN data plane enabled with DoS detection module. In addition, we implement network actions that can be utilized for running general middlebox functions, e.g., NAT (Network Address Translation) and ARP (Address Resolution Protocol) actions. Currently, DPX supports six network security actions and three additional network actions (summarized at Table 2).

We implement the prototype of DPX into two versions, i.e., software version and hardware using Open vSwitch Open vSwitch (2022); Pfaff et al. (2015) and NetFPGA-10G-SUME NetFPGA (2022); Zilberman et al. (2014) respectively to show that our work can be widely adopted in a real-world environment of both physical infrastructure and virtualized infrastructure. Also, our evaluation indicates DPX incurs a negligible overhead when compared to a naive forwarding switch and NFV, meaning that it can support line-rate of 10 Gbps and 0.5 ms of latency while providing security functions. In addition, we present several use-cases on how to detect and respond to network attacks with DPX's security actions. Our experiments highlight how DPX reduces complicated flow tables emanating from network service chains. In our scenarios, DPX successfully intercepts all attempted network attacks and compresses the number of required flow rules.

Contributions In summary, this paper makes the following contributions:

- We present the design of a new data plane architecture called DPX, which provides security services on the switch directly. It represents security services as a set of OpenFlow actions with optimized packet processing and simplified flow tables.
- We introduce *action clustering*, that logically integrates multiple DPX actions into a single action. This technique compresses complicated flow rules and eliminates unnecessary actions.
- We suggest the application of action clustering, i.e., advanced and global action clustering. They enable more sophisticated policies, and detect and mitigate a network-wide attack respectively.
- We implement and evaluate a prototype of DPX using Open vSwitch with six security actions and NetFPGA-10G-SUME with two security actions. Our evaluation indicates DPX incurs a negligible overhead when compared to a naive forwarding switch and NFV, meaning that it can support line-rate of network security services.

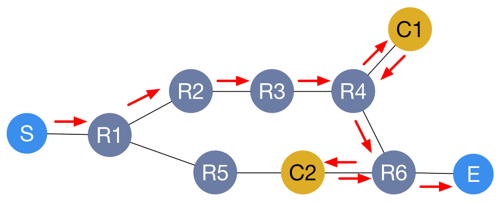
2. Background and motivation

This section presents the limitations of existing security solutions to motivate our work.

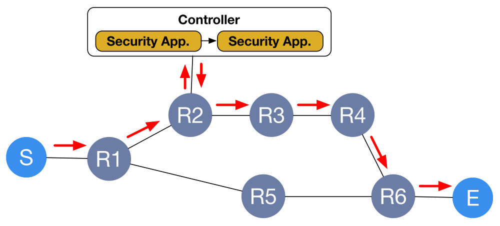
2.1. Limitations of existing security solutions

Although there has been a paradigm shift in the operation method in the modern network environment, various attacks on/over the network, such as DDoS attacks, scanning, or remote exploitation, are still effective in the dynamic network environment. Worse, as more services (e.g., Internet-of-Things, autonomous cars, cellular phones) are being deployed to the dynamic network, network threats are becoming more diverse, and the expected damage from attacks has increased than in the past (Antonakakis et al., 2017). Thus, security functions are an essential component in the dynamic network.

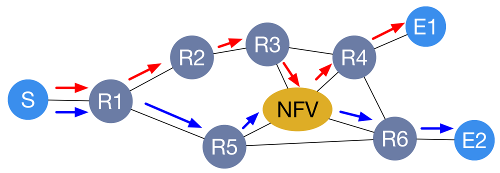
To protect dynamic networks from threats, we posit that network operators can take three types of security solutions: 1) Middlebox-based approach, 2) SDN-based approach, and 3) NFV-based approach (see Fig. 1). In what follows, we analyze each approach from the *performance* and *management* perspectives and discuss its limitations. **1) Middlebox-based approach** (Fig. 1(a)): A traditional approach is to deploy middleboxes at network choke-points and steer traffic through there by configuring forwarding rules (Fayazbakhsh et al., 2014; Gember-Jacobson et al., 2014; Liu et al., 2020; Qazi et al., 2013; Shin and Gu, 2012). As security operations (e.g., packet modification, filtering, inspection) are processed on dedicated hardware, its throughput outperforms other approaches.



(a) Middlebox-based approach [4] ($C1$ and $C2$ are choke-points where middleboxes are deployed.).



(b) SDN-based approach.



(c) NFV-based approach.

Fig. 1. The illustration of deployment strategies for security solutions in a dynamic network. S and E denote a source and destination respectively. $R*$ denotes network devices (i.e., routers or switches).

Yet, steering traffic through middleboxes leads to network-wide performance loss because specific bandwidth along the rerouting path should be reserved. Also, the longer the steering distance, the more latency users will experience. Last but not least, this approach requires significant costs to purchase middlebox devices or enforce complex forwarding rules.

2) SDN-based approach (Fig. 1(b)): SDN applications can implement security functions by utilizing a controller's global visibility and network-wide control (Kang et al., 2016; Lee et al., 2017; Yang et al., 2017; Yoon et al., 2015; Yu et al., 2017). These applications instruct SDN switches using the control channel (i.e., OpenFlow protocol McKeown et al., 2008) to perform diverse packet processing. In addition, considering that all network devices are connected to the controller in an SDN-enabled network, network operators can easily deploy security functions across the network without the need for traffic steering.

However, there is a significant performance overhead in that a single controller must handle all traffic in networks. According to Yoon et al. (2015), SDN security applications suffer significant performance degradation from several hundred Mbps to under tens of Mbps. Also, due to the limited features of the OpenFlow protocol, only a simple header-based inspection is supported (Shin et al., 2013). Therefore, SDN applications are not suitable to deploy practical security solutions. **3) NFV-based approach** (Fig. 1(c)): Deploying VMs that run software security applications (e.g., Snort, Suricata) is the widely adopted strategy by network operators, given its cost-efficiency and flexibility (Anderson et al., 2012; Bremler-Barr et al., 2016; Hwang et al., 2014; Sekar et al., 2012). With NFV, security functions can be placed in optimal locations close to routing paths, and designing a complex service chain (e.g., NAT-proxy-firewall) is also straightforward. Therefore, the NFV-based approach is now considered as the most promising solution for deploying security functions into the dynamic network environment.

Nevertheless, the NFV-based approach has a performance issue as traffic still needs to be steered through NFV nodes. More specifically, overall performance is significantly degraded if multiple NFV nodes are employed (Nam et al., 2018; Yu et al., 2015). To illustrate this, let us consider the example shown in Fig. 2. When deploying NFV-based security solutions, it is common to compose a service chain with a sequence of NFV nodes (i.e., VM1-VM2-VM3 in the example). Here, when a packet goes from an NFV node to another NFV node (or from the switch to an NFV node), each node should parse the packet to analyze its header. But, this causes all the nodes to perform duplicated tasks, which could be a key factor for performance degradation.

To verify this hypothesis, we conduct a simple benchmark. We measure the end-to-end throughput when the packet traverses at 10 Gbps speed to an NFV node that is connected with a software or hardware switch. The NFV node does nothing and returns traffic immediately when receiving the traffic. As shown in Fig. 3, the NFV node only achieves about 50% throughput than the baseline where no NFV node is deployed (i.e., `simple`). Thus, even if the NFV-based approach can deploy security functions to more optimal locations than the middlebox-based approach, steering traffic between NFV nodes causes performance degradation.

In addition, the NFV-based approach requires network operators to handle complicated policies, raising management complexity. This challenge is becoming more serious considering a security service chain that integrates multiple security functions in a series. For instance, let us consider the example shown in Fig. 4(a), that configures five flows with different service chains. In order to operate these service chains, complicated flow rules are required in SDN switches to forward packets between node-to-node or switch-to-nodes (see Fig. 4(b)). Eventually, the switches' flow table would be complicated and messy. Further, a network operator should manage extra control channels for each NFV node (i.e., the NFV orchestrator in Fig. 2). This is opposite with the design philosophy of SDN, which operates a network from a centralized location.

2.2. Our solution

DPX eliminates the need for traffic steering through making SDN switches support native security functions. By doing so, network operators can directly implement security functions on switches without deploying additional middleboxes or NFV nodes. This approach avoids performance degradation because traffic no longer needs to be steered through other locations. In addition, DPX provides a variety of practical security functions to support the same level of security features with NFV. Specifically, DPX extends OpenFlow to incorporate security features that perform advanced actions, such as payload inspection, rate detection, scanning

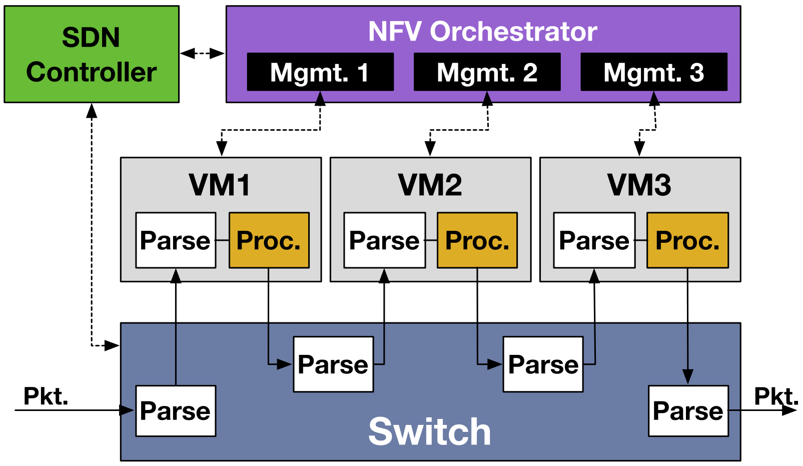
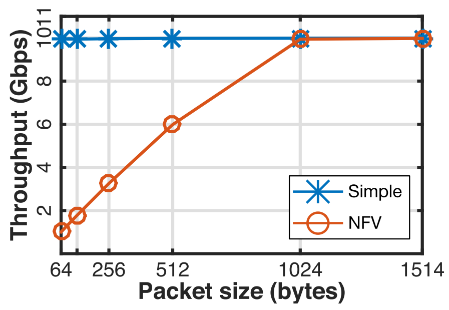
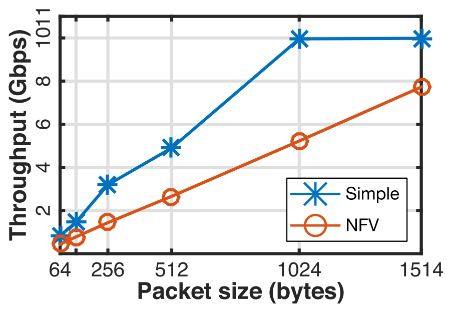


Fig. 2. NFV operation breakdown.

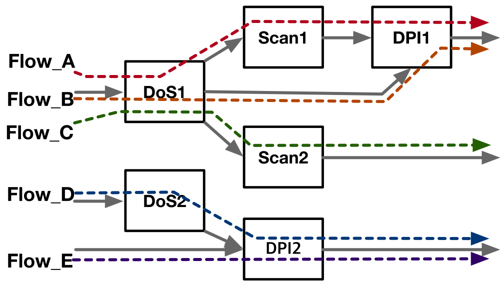


(a) NFV with hardware switch.

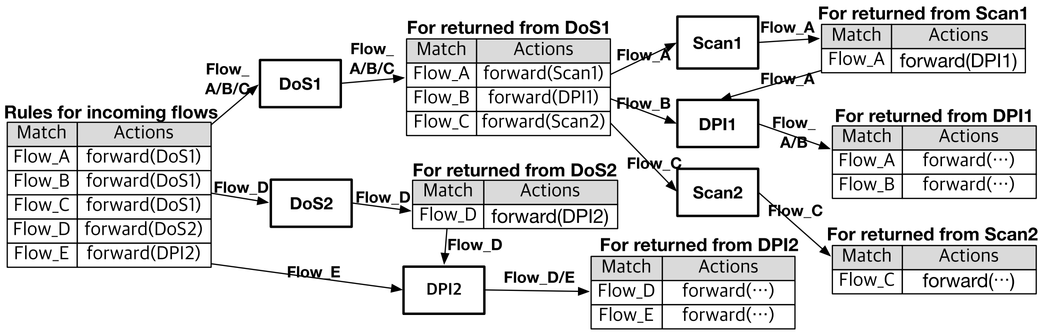


(b) NFV with software switch.

Fig. 3. Performance degradation caused by traffic steering.



(a) An example of security service chains on NFV nodes.



Rules for incoming flows

Match	Actions
Flow_A	forward(DoS1)
Flow_B	forward(DoS1)
Flow_C	forward(DoS1)
Flow_D	forward(DoS2)
Flow_E	forward(DPI2)

For returned from DoS1

Match	Actions
Flow_A	forward(Scan1)
Flow_B	forward(DPI1)
Flow_C	forward(Scan2)

For returned from DoS2

Match	Actions
Flow_D	forward(DPI2)

For returned from Scan1

Match	Actions
Flow_A	forward(DPI1)

For returned from DPI1

Match	Actions
Flow_A	forward(⋯)
Flow_B	forward(⋯)

For returned from Scan2

Match	Actions
Flow_C	forward(⋯)

For returned from DPI2

Match	Actions
Flow_D	forward(⋯)
Flow_E	forward(⋯)

(b) Flow rules installed in SDN switches to handle the service chains.

Fig. 4. An illustration of the management challenge in security service chaining.

detection, header modification, and session monitoring. As these functions are supported in SDN switches, network operators do not need to manage another control channel (i.e., NFV orchestrator) and additional security nodes in SDN-enabled networks.

3. Related work

This section introduces related works around enhancing network security with software solutions. Table 1 summarizes the comparison of DPX and the SDN switch extension and programmable-switch-based solutions.

Network function control Prior research explores the possibility of controlling network functions using middleboxes. CloudWatcher (Shin and Gu, 2012) uses OpenFlow to detour network flows to physical network devices in dynamic cloud networks, where the security functions are pre-installed. SIMPLE Qazi et al. (2013) and Gupta et al. (2018) propose the efficient traffic steering for composing service chaining with middleboxes. Whereas these systems streamline NFV deployment, they do not address the performance degradation challenge due to the need for traffic steering.

Software switches Existing works on software switches mostly focus on improving system performance (Honda et al., 2015; Intel, 2022a; Pfaff et al., 2015). Popular software switch implementations are Open vSwitch (2022); Pfaff et al. (2015) and mSwitch (Honda et al., 2015). They aim to achieve high-performance by supporting a large number of virtual ports. CoMb (Sekar et al., 2012) consolidates network middleboxes into a single physical machine to reduce capital expenses and device sprawl. It also designs consolidated protocol parsers to optimize and reduce repeated packet parsing steps between diverse network functions.

Offloading network functions Offloading network functions into a low-level network stack (e.g., SmartNICs or FPGA) is being used popularly in data centers to achieve high-performance. AccelNet (Firestone et al., 2018) proposes the FPGA-based SmartNIC to handle mass bandwidth on their network center. Mobius (Park and Shin, 2021) suggests a rich network policy handling way in the hardware-based network data plane. Reinhardt (Park et al., 2021) proposes a reconfigurable FPGA architecture tailored for payload inspection. Inspired by those projects, we aim to design the extended data plane (i.e., switch) that specifically addresses the needs of security functions and reduces latency associated with service chains.

Security with SDN switch extension Like our idea, several works propose an extended SDN data plane to support rich functions from an OpenFlow switch. OFX (Sonchack et al., 2016a) is an OpenFlow extension framework which enables an security application to be loaded into a switch at runtime. NEWS (Mekky et al., 2017) suggests an extended SDN architecture to handle packets through modified flow tables on a switch, called app tables. AvantGuard (Shin et al., 2013) proposes a secure OpenFlow switch architecture that supports the connection migration and actuating triggers to enhance the scalability and responsiveness of OpenFlow switches. QoSE (Park et al., 2016) proposes a data plane module for providing security features as distributed NFV. Whereas those works are similar to us, DPX incorporates all the security functions the existing works propose and supports additional advanced ones, including network functions.

Security with programmable switches Recently, diverse switch-native solutions have been presented by utilizing programmable data planes (e.g., P4-enabled switches). For example, Poseidon (Zhang et al., 2020) and Jaqen (Liu et al., 2021) propose a switch-native approach for mitigating volumetric DDoS attacks, leveraging the reconfigurability and processing power of switch devices. iMap (Li et al., 2022) proposes a network scanner that performs large-scale scanning in a fast and scalable manner. While the idea is similar to our work, their solutions require programmable ASICs (e.g., Intel Tofino Intel, 2022b) and a domain-specific language (e.g., P4 Bosshart et al., 2014) different from OpenFlow. Considering that SDN is still widely used in data centers (Ferguson et al., 2021), WAN (Hong et al., 2018), and enterprise networks (BlueCat Networks, 2022), DPX can be readily deployed to existing SDN switches. Also, the programmable-switch-based solutions do not support payload inspection, while DPX does so.

4. System design

In this section, we present the design of DPX. We first provide a DPX architectural overview, and introduce DPX actions that are designed for supporting security functions from SDN switches. We then introduce action clustering, a novel technique to relax management complexity of the extended security actions.

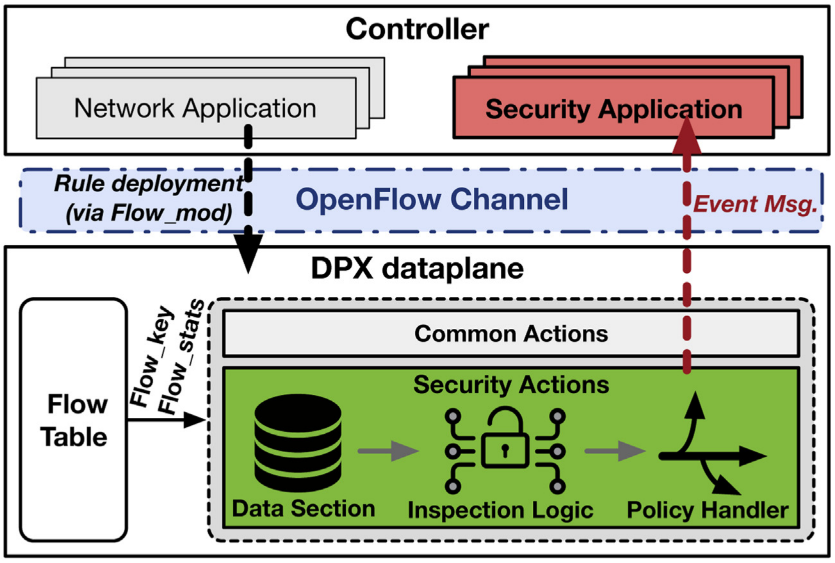
4.1. DPX Overview

Fig. 5 illustrates the overall design of DPX and its workflow. DPX follows the OpenFlow protocol paradigm (OpenFlow, 2022) that handles network packets using the match-action interface of an SDN switch. Thus, DPX extends the OpenFlow implementation of SDN switches to incorporate *security actions* as an additional set of OpenFlow actions. By doing so, the applications running on the *controller* can enforce security functions on network traffic to DPX data-plane via the extended OpenFlow protocol.

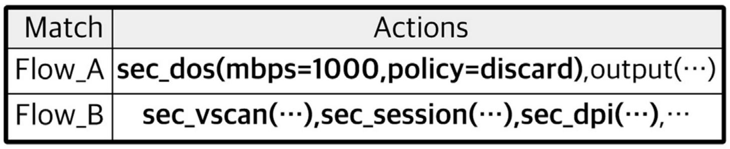
In DPX, security functions are defined in terms of one or more actions that the data plane provides to handle network traffic (e.g., packet forwarding, drop, and modification). For instance, in the flow table of Fig. 5(b), the actions for `Flow_A` will monitor network flows to detect whether flows send/receive more than 1000 Mbps threshold, and if it detects this situation DPX will conduct pre-defined operations (e.g., generate alert or drop packets). In addition, the actions for `Flow_B` will perform multiple network security functions (i.e., vertical scan detector, session monitor, and deep packet inspector) to the corresponding traffic. Since DPX builds on top of the OpenFlow protocol, security actions can be enforced with the existing OpenFlow `FLOW_MOD` commands and any other OpenFlow actions can be integrated with DPX actions.

Table 1
Comparison of DPX and the existing works (✓Fully applied, △ Partially applied, ✗Not applied.) .

Work	Purpose	No Traffic detouring	Compatibility with SDN	Payload inspection
Poseidon (Zhang et al., 2020)	DDoS mitigation	✓	✗	✗
Jaqen (Liu et al., 2021)	DDoS mitigation	✓	✗	✗
Ripple (Xing et al., 2021)	DDoS mitigation	✓	✗	✗
Avant-Guard (Shin et al., 2013)	DDoS mitigation	✓	✓	✗
QoSE (Park et al., 2016)	Traffic steering optimization	✗	✓	✗
SIMPLE (Qazi et al., 2013)	Traffic steering optimization	✗	✓	✗
NEWS (Mekky et al., 2017)	Dataplane extension	✓	✓	✗
OFX (Sonchack et al., 2016a)	Dataplane extension	△	✓	✗
DPX	Dataplane extension	✓	✓	✓



(a) DPX system architecture.



Match	Actions
Flow_A	**sec_dos(mbps=1000,policy=discard)**,output(⋯)
Flow_B	**sec_vscan(⋯),sec_session(⋯),sec_dpi(⋯)**,⋯

(b) Example flow table.

Fig. 5. An illustration of DPX design and processing workflow.

4.2. DPX actions

A DPX action operates in a similar way to the match-and-action sequence of OpenFlow; after looking up a matched flow rule in flow tables for incoming packets (i.e., parsing packets, looking up flow entries corresponding to the packets, and updating flow statistics), DPX executes the actions field in the matched rule. If an action field includes (a) security action(s) during execution, DPX will trigger a corresponding DPX action block to provide security functions for matched flows. As shown in Fig. 5(a), the blocks in the security action comprise *data section, inspection logic*, and *policy handler*. With those blocks, the DPX performs the following three stages:

First, DPX reactively/proactively updates its entries with keys according to security action types. The data section is responsible for keeping the context of security actions. It consists of (1) *flow_key*, the packet-level metadata used for indexing flow tables, and (2) *flow_stats*, the statistical data of each flow entry (e.g., packet counts, bytes). For example, the action for DoS detection updates the size of an incoming packet and its arrival time, and the action for scanning detection updates the last access time and list of accessed TCP/UDP ports. On the other hand, the action for deep-packet inspection (DPI) requires a pattern list to match its entries with packet payloads. For this, DPX provides a user-defined pattern list to the data section at initialization.

Second, DPX performs the inspection logic, which refers to the packet inspection operations that need to be executed for a security function. For instance, a DoS detector calculates bps (bits-per-second) of a flow using the metadata and statistical information for this flow (i.e., size and time of packets in its data section). Also, a scanning detector counts how many ports are hit within a time window using the last access time and port list in the data section. After executing the inspection logic, its result is compared with the condition value set by action's parameters to decide whether or not packets violate the security specification.

Third, if a security violation is detected by the inspection logic, DPX handles packets according to one of four policies: (1) `neglect` which ignores the events and processes packets following a normal switch sequence; (2) `alert` which sends an alert message to a controller with a switch ID (i.e., datapath ID), physical port number associated, the reason for event occurrence (event type code), reference features (e.g., the current bps), raw packet data, and a cluster ID (We will describe this in the next section.); (3) `discard` which terminates the packet processing sequence and drops detected packets; and (4) `redirect` which forwards packets to alternative destinations (e.g., honeypot) instead of the original destination specified in a flow rule.

Network operators can configure DPX actions via parameters like common OpenFlow actions (e.g., `set_nw_src(10.0.0.1)` that modify the source IP address to 10.0.0.1). The parameters are defined as two types: *feature variables* and *policy*. The feature variables indicate trigger conditions or configuration values. For example, the bps threshold and the pattern list are the feature variables for the DoS detector and the deep-packet inspector, respectively. Depending on the type of security actions, there may be one or more feature variables. The policy dictates how the detected packets are handled. Specifically, operators can specify the DDoS attack mitigation policy "if 1000 Mbps traffic is detected, redirect it to port 2" as "`sec_dos(mbps=1000,policy=redirect:2)`".

DPX supports six security actions (see the top of Table 2). Each action has different feature variables related to its purpose, and the feature variables must be set when a security action is installed. The policy can be omitted, and an alert is set as default when the policy is not set. As mentioned before, a DPX action consists of multiple packet processing blocks. Thus, it is possible to compose a new action by combining different blocks. Therefore, more func-

Table 2
DPX actions overview.

Function	Action name	Purpose Feature variable	Description example
Deep Packet Inspector	`sec_dpi`	Find a pattern in a packet payload. - *rule:* The path of a rule file containing patterns	`sec_dpi(rule=` *"pattern_list.txt"*, `policy=alert)`
DoS Detector	`sec_dos`	Detect a bandwidth exceed by mbps threshold. - *mbps:* The bandwidth threshold to detection	`sec_dos(mbps=`*1000*, `policy=alert)`
Anomaly Detector	`sec_anomaly`	Detect a change rate of bandwidth within the buffer interval. - *delta:* The percentage of the change rate threshold for bandwidth - *buf:* The packet buffer length to recognize the change rate	`sec_anomaly(delta=`*200*, `buf=`*1024*, `policy=alert)`
Vertical-Scanning Detector	`sec_vscan`	Count how many TCP/UDP ports are hit within a time window. - *ports:* The count threshold of hit ports - *time:* The time window size for the ports count	`sec_vscan(ports=`*1000*, `time=`*5*, `policy=alert)`
Horizontal-Scanning Detector	`sec_hscan`	Count how many hosts for the specific TCP/UDP port are hit within a time window. - *hosts:* The count threshold of hit hosts - *time:* The time window size for the hosts count - *port:* The target TCP/UDP port number to monitor	`sec_hscan(hosts=`*100*, `port=`*80*, `time=`*5*, `policy=alert)`
Session Monitor	`sec_session`	Trace TCP sequences and count invalid connections. - *session:* The count threshold of invalid sessions - *time:* The time window size for the session count	`sec_session(count=`*100*, `time=`*10*, `policy=alert)`
Load-Balancer	`srv_load`	Distribute network traffic across a number of ports. - *ports:* The target output port list to distribute traffic	`srv_load(ports=`*1,2,3,4*, `policy=neglect)`
Network Address Translator (NAT)	`srv_nat`	Remap IP addresses into another one and forward to a port. - *ip:* The IP address to change for a flow - *output:* The target output port	`srv_nat(ip=`*'10.0.0.1'*, `outport=`*1*, `policy=neglect)`
Address Resolution Protocol (ARP)	`srv_arp`	Map IP addresses to hardware addresses. - *arp:* The list of IP to MAC address mappings.	`srv_arp(arp=` *"10.0.0.1-aa:bb:cc:dd:ee:ff, 10.0.0.2-01:02:03:04:05:06"*, `policy=neglect)`

tions beyond the six security actions can be easily designed in the current DPX architecture. For example, network operators can design middlebox functions, such as load-balancer, network address translator, and address resolution protocol handler (see the bottom of Table 2).

Benefits of DPX *actions* DPX actions provide significant benefits over traditional middlebox or SDN/NFV-based security solutions:

1) *Simplified policy management.* DPX actions enable to achieve operational efficiency by eliminating the need for installing additional flow rules for traffic steering. Thus, a network operator can focus on the management of existing flow rules (e.g., whether normal traffic reaches its destination).
2) *Simplified service chaining.* DPX actions enable to compose a service chain in a simplified manner. Specifically, a single line of flow rule is sufficient to represent a complex service chain by enumerating multiple actions. For example, suppose we configure a service chain for a DoS detector, anomaly detector, vertical-scanning detectors, session monitor, and payload inspector of a flow destined to a 10.0.0.1 host. In that case, it can be expressed as a single rule as follows:
`Flow`:..., `nw_dst`=10.0.0.1,..., **actions=sec_dos**(...), **sec_anomaly** (...), **sec_vscan**(...), **sec_session**(...), `sec_dpi`(...),...
3) *Simplified processing sequence.* DPX enables to improve network performance by minimizing packet processing steps for security functions. As depicted in Fig. 6, processing an incoming packet generally requires four steps (i.e., parse packets, look up flow tables, update flow stats, and execute actions). Whereas the NFV-based approach requires to perform this twice due to the traffic steering through the NFV node (i.e., `Flow_A`), DPX only needs a single processing sequence because a switch can fully support security functions (i.e., `Flow_B`).

4.3. Action clustering

Although DPX actions eliminate the need for traffic steering, there remains an additional challenge due to the limited OpenFlow design. When configuring rules to implement advanced security actions (e.g., counting the number of packets for selected rules), it is common to utilize the OpenFlow multi-table feature,[1] allowing a switch to compose packet processing pipeline. To illustrate this, let us consider Fig. 7 where a network operator wants to design a DoS detector by aggregating packets of `Flow_A`, `B` and `C`. Starting from the default table (`Table 0`), those flows are redirected to the Table 1 to execute the `sec_dos` action, and then forwarded to the Table 2 to be separated into the remaining service chains. Whereas it enables network operators to design service chains *within* a switch, operators should take care of installing rules in multiple tables, increasing management complexity.

In order to address this problem, we propose a novel technique called *action clustering*. Its purpose is to simplify rule complexity by merging the DPX actions into few synthetic rules. Thus, the same DPX action can be executed across flow rules without redundant packet processing, i.e., flow aggregation and separation. This way, both the number of flow rules and processing time can be significantly reduced than the case when OpenFlow multi-table features are used.

Fig. 8 illustrates the workflow of action clustering. Each DPX action is assigned a cluster ID that groups multiple DPX actions into the same cluster. DPX then builds the clustering map per action type and maintains the aggregated data per cluster ID. For example, `Flow_A` and `Flow_C` have the same action type `sec_dos` and cluster ID `10`. When the packets of `Flow_A` and `Flow_C` arrive at a switch, the aggregated statistics (e.g., bytes per second) are updated in the clustering map of the `sec_dos` action. On the other hand, if a DPX action runs standalone without clustering, its cluster ID is set to a unique random ID (see `Flow_B`). Also, DPX considers a {`action type, cluster ID`} pair as a unique key;

[1] Note that some OpenFlow switches still do not support the multi-table feature at the time of writing Pica (2022).

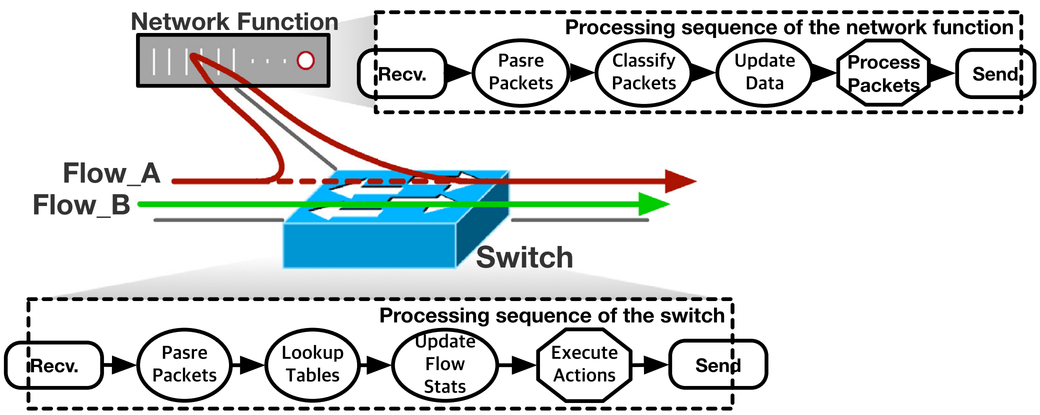


Fig. 6. Inefficient NFV packet processing sequence.

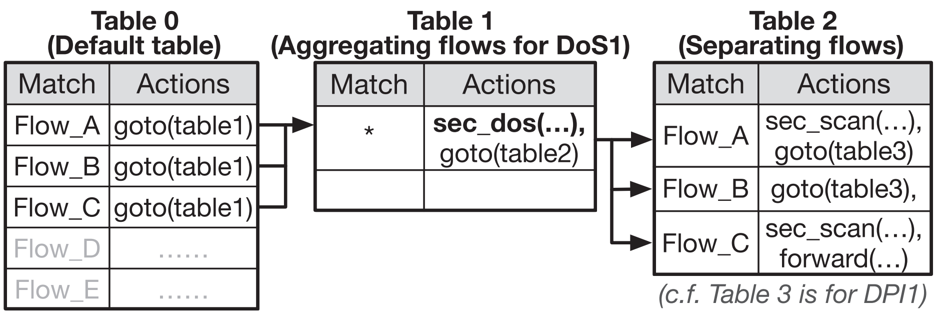


Table 0 (Default table)
Match | Actions
Flow_A | goto(table1)
Flow_B | goto(table1)
Flow_C | goto(table1)
Flow_D | ……
Flow_E | ……
Table 1 (Aggregating flows for DoS1)
Match | Actions
* | sec_dos(…), goto(table2)
Table 2 (Separating flows)
Match | Actions
Flow_A | sec_scan(…), goto(table3)
Flow_B | goto(table3),
Flow_C | sec_scan(…), forward(…)
(c.f. Table 3 is for DPI1)

Fig. 7. An example of service chaining with the OpenFlow multi-table feature.

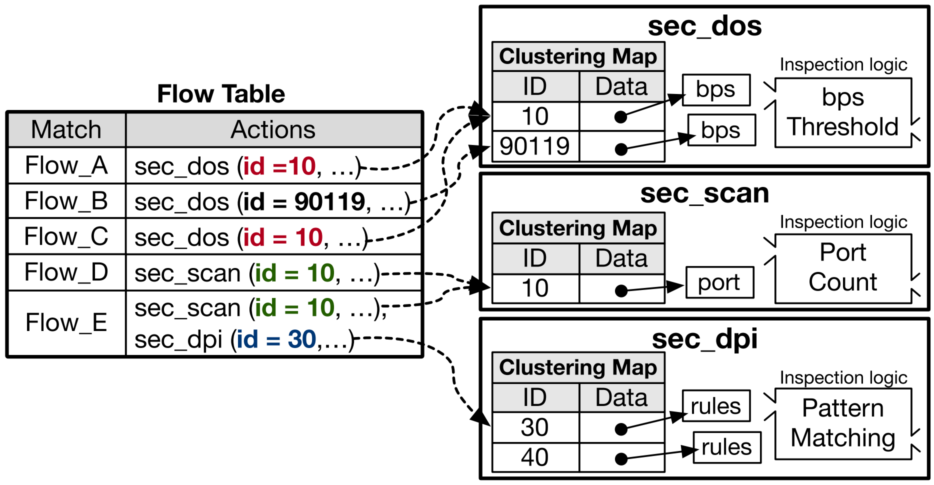


Fig. 8. An example of service chaining using DPX action clustering.

thus, the actions having the same cluster ID are differentiated according to its type, e.g., the `sec_dos` action for `Flow_A/C` and `sec_scan` action for `Flow_D`. The action clustering operates independently for all actions in a flow rule (see `Flow_E`).

The cluster ID is used as the hash key to lookup the clustering map, and the *data* referenced by the cluster is updated and delivered to the inspection logic. The data section is shared between different flows so that the input of one flow can affect the inspection result of another flow within the same cluster. Therefore, we can detect not only the abnormal behavior of aggregated flows but also individual ones.

4.4. Advanced action clustering

To make the action clustering more flexible in practice, DPX supports two additional features, *inconsistent-parameter clustering* and *multi-clustering*.

Inconsistent-parameter clustering In some circumstances, the parameters of actions within the same cluster can be different. For this case, we design inconsistent-parameter clustering, which executes a processing logic with the parameter of each action regardless of the cluster it belongs. The following flow rules are examples of inconsistent-parameter clustering:

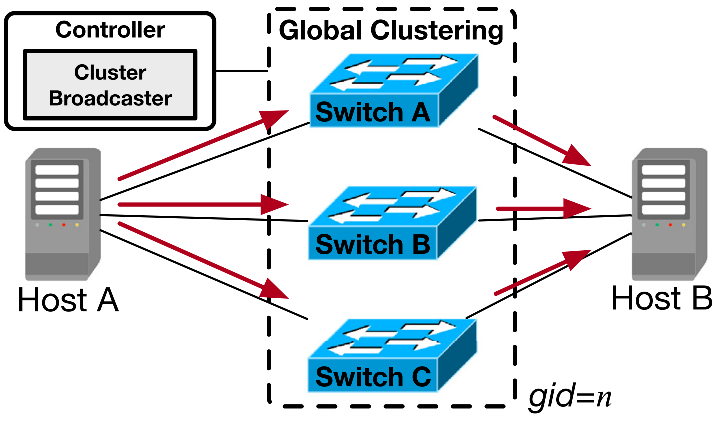


Fig. 9. Design of global action clustering.

`Flow_A: actions=sec_vscan(`**ports=1000,time=5**`,id=10),...`
`Flow_B: actions=sec_vscan(`**ports=500,time=3**`,id=10),...`

The `sec_vscan` action detects a vertical scanning attack by counting how many ports are hit within a specific time window. Since the `sec_vscan` actions belong to the same cluster, they share the same data that contains the number of port hits and the last arrival time of each port. However, each `sec_vscan` has different parameters for detecting scanning attacks, meaning that multiple security policies are applied to the same data. For instance, let us suppose that the current aggregated number of port hits is 700 during the last three seconds and 900 during the last five seconds. In that case, DPX only triggers an alert against `Flow_B`, not `Flow_A`.

Multi-clustering A DPX action can be assigned to multiple clusters for computing different statistics. When DPX executes an action containing multiple cluster IDs, it updates the data for all the clusters. Thus, the multi-clustering allows network operators to enforce multiple security policies for a single flow. The following flow rules are an example.

`Flow_A: actions=sec_dos(mbps=1000,`**id=10**`),...`
`Flow_B: actions=sec_dos(mbps=500,`**id=10,20**`),...`

The `sec_dos` action for `Flow_B` is assigned two cluster IDs, 10 and 20, while `Flow_A` is only assigned in one cluster ID 10. This implies that the action of `Flow_B` monitors `mbps` along with `Flow_A`. By doing so, network operators can set complex security policies such as “`Flow_A` and B should not exceed 1000 Mbps” or “`Flow_B` should not exceed 500 Mbps (But, `Flow_A` can reach 1000 Mbps.)”.

4.5. Global action clustering

As DPX operates security functions on each switch locally, DPX has difficulty in providing network-wide security solutions that require global coordination across a network (e.g., botnet-driven DDoS attacks). To address this challenge, we design a data-exchange protocol between switches, called *global action clustering*. It aims to achieve network-wide policy enforcement by exchanging the data section of a DPX action.

Fig. 9 illustrates the design of global action clustering. Each DPX action can optionally have a global cluster ID (i.e., `gid`), grouping DPX actions of different switches into the same cluster. The grouped switches broadcast their local data through the *cluster broadcaster* module on the controller so that they can synchronize the data with each other. The data broadcast from other switches is managed in the *global cluster map* of the data section, independently of the local cluster map (see Fig. 10). If an action includes a `gid`, DPX looks up not only the local cluster map, but the global one to execute the inspection logic.

Table 3
An example of global action clustering for detecting DDoS attacks.

sec_dos(gid = n,mbps = 1000)	Switch A	Switch B	Switch C
Local	300	350	450
Global	800	750	650
Local/Global ratio	0.375	0.466	0.692

Table 3 is an example to show how the global clustering works in the topology of Fig. 9 under DDoS attacks. Let us suppose that the switch A, B and C belong to the global cluster n and want to detect 1000 Mbps traffic. Each switch receives 300, 350, and 450 Mbps of traffic, and these data are broadcast to the neighbor switches in the cluster. The global cluster value of each switch is the sum of the local values of other two switches, i.e., 800 (350 + 450), 750 (300 + 450) and 650 (300 + 350), respectively. Suppose the sum of local and global values exceeds the threshold (i.e., 1000). In that case, its policy is applied when a local/global ratio exceeds 0.5, meaning that the switch is receiving more traffic than the other two switches. In the example, all switches exceed 1000 Mbps, so DPX checks its local/global ratio; switch A is 0.375 (300/800); B is 0.466 (350/750); and C is 0.692 (450/650). Therefore, only switch C executes inspection logic for this case. For example, if the policy of switch C is `alert`, the controller receives an alert message from switch C with the global ID n.

4.6. Programming DPX applications

DPX helps network operators design security policies easily by following the OpenFlow programming convention. Thus, network operators can apply DPX on their SDN networks without learning a new language. The only additional bit of information is the specific parameters required for certain DPX actions. By extending the existing OpenFlow protocol to utilize DPX features, network operators can easily develop DPX applications for automatic security management with minimal effort.

Fig. 11 shows an example DPX application, which deploys a DoS detector and DPI function to a switch by appending `sec_dos` and `sec_dpi` actions into an OpenFlow `FLOW_MOD` message (lines 1–7). When the switch detects a packet whose pattern is matched with `pattern.txt` for the `alert` policy, it sends the SDN controller an OpenFlow message. Then, the DPX application responds with suitable reactions to abnormal behaviors by invoking its event handlers (lines 8–13).

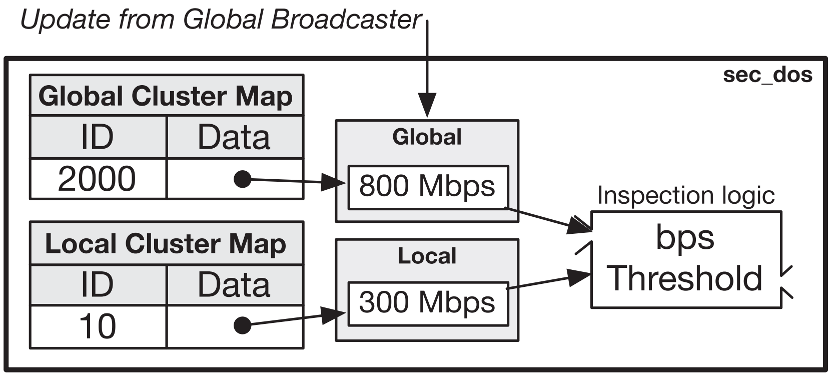


Fig. 10. Global and local cluster maps.

```
1: procedure DeployDPXSecurityActions
2:     $msg \leftarrow$ openflow.flow_mod()
3:     $msg.match.in\_port \leftarrow 1$
4:     $msg.actions[0] \leftarrow$ DPX.sec_dos(mbps=1000,id=10,policy=redirect:2)
5:     $msg.actions[1] \leftarrow$ DPX.sec_dpi(rule=pattern.txt,id=20,policy=alert)
6:     $msg.actions[2] \leftarrow$ openflow.action.output(2)
7:     send_to_controller($msg$)
8: procedure DPXMessageHandler($alert$)
9:     $packet \leftarrow$ event.packet                    ▷ The parsed packet data
10:    print "in_port: $alert.in\_port$"
11:    print "alert_reason: $alert.reason$"
12:    print "cluster_id: $alert.cluster\_id$"
13:    print "value: $alert.value$"
```

Fig. 11. DPX security application pseudo code.

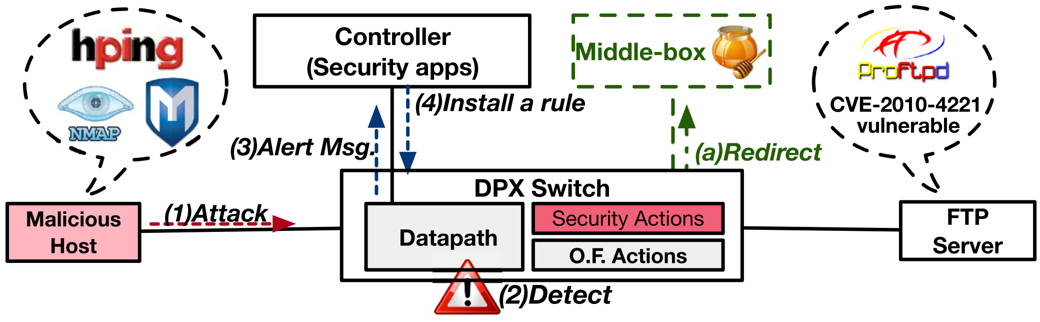


Fig. 12. Testbed and operational scenario of DPX use cases.

5. Security use cases

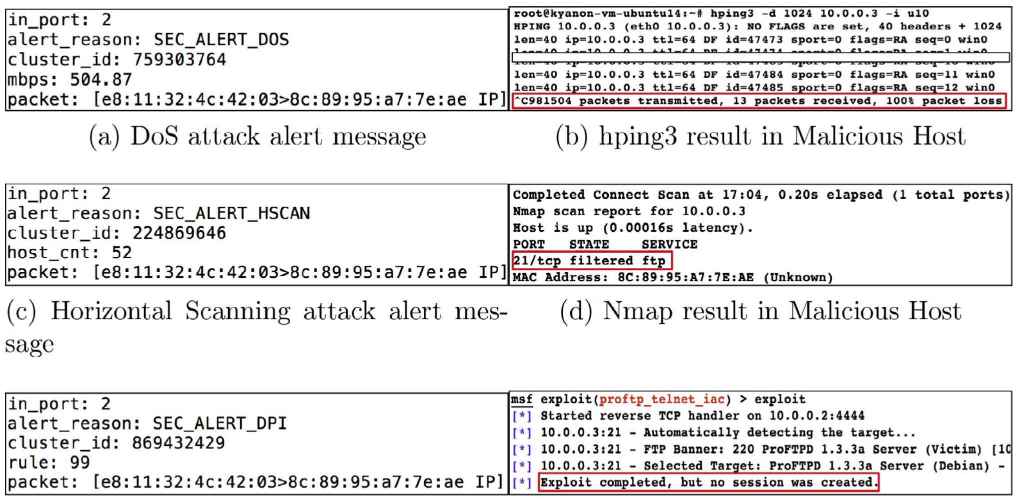
This section presents various security use cases of DPX.

5.1. Security inspection

To highlight operational use-cases of DPX, we set up a testbed (shown in Fig. 12), where a DPX switch connects two hosts (one benign and one malicious) and a server (hosting FTP service, which has a buffer-overflow vulnerability). The switch is connected to a controller running DPX security applications. The malicious host performs three different attacks: (i) DoS, (ii) port scanning, and (iii) remote exploit against the FTP server. Here, we will present how the DPX switch analyzes ongoing network traffic, detects attacks, and reports to the controller. We assume that a network administrator has pre-configured the security applications for reacting to network attacks.

Denial of service attacks In this example, we employ the `sec_dos` action to alert when the traffic surpasses 500 Mbps and the malicious host sends over 1 Gbps traffic to the FTP server using `hping3` (HPING3, 2022). When the `sec_dos` action identifies the attack (i.e., high-volume traffic), it sends an alert message including current packet-rate information to the DPX controller (Fig. 13(a)). Then, the DPX application installs a new flow rule to block the attack traffic in DPX. Hence, most of the traffic is dropped, and the DoS attack is mitigated as shown in Fig. 13(b).

Port scanning DPX can detect horizontal and vertical scanning via its scan detector actions, such as `sec_vscan` and `sec_hscan`. In addition, the `sec_session` action can help in detecting stealthy scanning attacks. In this example, we configure the `sec_hscan` with 50 hosts and a 10-second time window. Next, the malicious host generates horizontal scans directed at port TCP/21 using `nmap` (NMAP, 2022). When DPX successfully detects the horizontal scanning, DPX sends an alert message including



```
in_port: 2
alert_reason: SEC_ALERT_DOS
cluster_id: 759303764
mbps: 504.87
packet: [e8:11:32:4c:42:03>8c:89:95:a7:7e:ae IP]
```

(a) DoS attack alert message

```
root@kyanon-vm-ubuntu14:~# hping3 -d 1024 10.0.0.3 -i u10
HPING 10.0.0.3 (eth0 10.0.0.3): NO FLAGS are set, 40 headers + 1024
len=40 ip=10.0.0.3 ttl=64 DF id=47473 sport=0 flags=RA seq=0 win0
len=40 ip=10.0.0.3 ttl=64 DF id=47474 sport=0 flags=RA seq=1 win0
len=40 ip=10.0.0.3 ttl=64 DF id=47483 sport=0 flags=RA seq=10 win0
len=40 ip=10.0.0.3 ttl=64 DF id=47484 sport=0 flags=RA seq=11 win0
len=40 ip=10.0.0.3 ttl=64 DF id=47485 sport=0 flags=RA seq=12 win0
^C981504 packets transmitted, 13 packets received, 100% packet loss
```

(b) hping3 result in Malicious Host

```
in_port: 2
alert_reason: SEC_ALERT_HSCAN
cluster_id: 224869646
host_cnt: 52
packet: [e8:11:32:4c:42:03>8c:89:95:a7:7e:ae IP]
```

(c) Horizontal Scanning attack alert message

```
Completed Connect Scan at 17:04, 0.20s elapsed (1 total ports)
Nmap scan report for 10.0.0.3
Host is up (0.00016s latency).
PORT    STATE    SERVICE
21/tcp filtered ftp
MAC Address: 8C:89:95:A7:7E:AE (Unknown)
```

(d) Nmap result in Malicious Host

```
in_port: 2
alert_reason: SEC_ALERT_DPI
cluster_id: 869432429
rule: 99
packet: [e8:11:32:4c:42:03>8c:89:95:a7:7e:ae IP]
```

(e) Remote exploit attack alert message

```
msf exploit(proftp_telnet_iac) > exploit
[*] Started reverse TCP handler on 10.0.0.2:4444
[*] 10.0.0.3:21 - Automatically detecting the target...
[*] 10.0.0.3:21 - FTP Banner: 220 ProFTPD 1.3.3a Server (Victim) [10
[*] 10.0.0.3:21 - Selected Target: ProFTPD 1.3.3a Server (Debian) -
[*] Exploit completed, but no session was created.
```

(f) Metasploit result in Malicious Host

Fig. 13. Alert messages & block results for various use-case attacks.

current host count to the controller (Fig. 13(c)). Then, the DPX application installs a new flow rule to block the attack traffic, as shown in Fig. 13(d).

Remote exploit We consider the case where the malicious host tries to exploit the vulnerability of ProFTPD to get a remote shell. We reproduce this attack via `Metasploit` (Metasploit, 2022), the most popular penetration testing tool, and set the `sec_dpi` action with 100 rules including the attack pattern at 99th. After performing the attack, the `sec_dpi` action detects the attack pattern in the packet payloads and issues an alert message to the controller. The alert indicates the corresponding pattern number that is matched in the pattern list (Fig. 13(e)). Then, the DPX application installs a new flow rule to block the attack traffic, preventing the malicious host from acquiring a remote shell through the exploit sequence, as shown in Fig. 13(f).

Cooperation with middleboxes DPX can cooperate with middleboxes using a `redirect` policy so that middleboxes can process suspicious connections. For example, the network in Fig. 12 runs a honeypot with security actions on the DPX switch. The security actions have the *redirect* policy, so benign connections are forwarded to the original destination, but suspicious connections are transmitted to the honeypot. This cooperation facilitates the implementation of other network security systems, such as reflectornets (Shin et al., 2016), and Moving Target Defense (MTD) (Kampanakis et al., 2014).

Security control solution Since DPX is designed to be compatible with OpenFlow (OF), a security application can be implemented on a controller by combining the capability of OF and DPX security features. Fig. 14 is the example application that is built on the POX controller. This application collects information about the switches by requesting OF statistics messages, such as `OFPC_FLOW_STATS`, `OFPC_TABLE_STATS` and `OFPC_PORT_STATS`, and monitors the status of deployed security actions through the DPX security handler. Therefore, by using the collected information, it can display the direction and amount of traffic between each switch and the current security status of switches. If a security violation occurs, an administrator can establish a security policy based on the observed network conditions. For example, in the case of Fig. 14, the `sec_anomaly` action of the switch `s3` alerts that current traffic-level is 285% higher than usual. The administrator could analyze the cause of this alert from the displayed traffic information, and determine that the switch `s4` is currently generating a large amount of traffic to `s3`. As a result, the administrator can block traffic for `s4` to `s3`, or deploy stricter security actions to defend against future attacks.

5.2. Resource-aware action deployment

Since DPX operates on OF switches, DPX can be rapidly deployed anywhere in an SDN network without imposing additional overhead. Leveraging this benefit, we present a *resource-aware action deployment* that adjusts a routing path considering network loads. Fig. 15 describes its workflow.

On the control plane, the *network monitor* module monitors information about link failures with Link Layer Discovery Protocol (LLDP) and bandwidth usage through the OF statistics messages. This information is used in the *security load distribution control* module for conducting optimization to derive a new routing path, where security functions are relocated. For example, in Fig. 15, the shortest path from the source (S) to the destination (E) is 'R1-R6-R7', but the routing path taking into account remaining resources is configured to 'R1-R2-R3-R5-R7'.

To conduct resource optimization, we leverage the formula of QoSE (Park et al., 2016), a distributed NFV system that finds optimal NFV node placement given current traffic amount. It assumes that the optimal resource distribution can be derived by calculating the amount of traffic allocated for each node.

We extend it to a general network model to find an optimal routing path. The resource capacity of nth switch on a single routing path is denoted by r_n, and an ith security action on the nth switch is denoted by $f_{n,i}$. For example, we mark $f_{2,1}$ to denote the first security action on the second switch of a path. The movement of traffic from a switch m to a switch n is marked by $b_{m,n}$. When the allocated traffic amount on $f_{n,i}$ is $b_{n,i}$, the resource consumption is marked by $f^r_{n,i}(b_{n,i})$. To this end, we formulate an integer linear programming (ILP) as follows:

Maximize

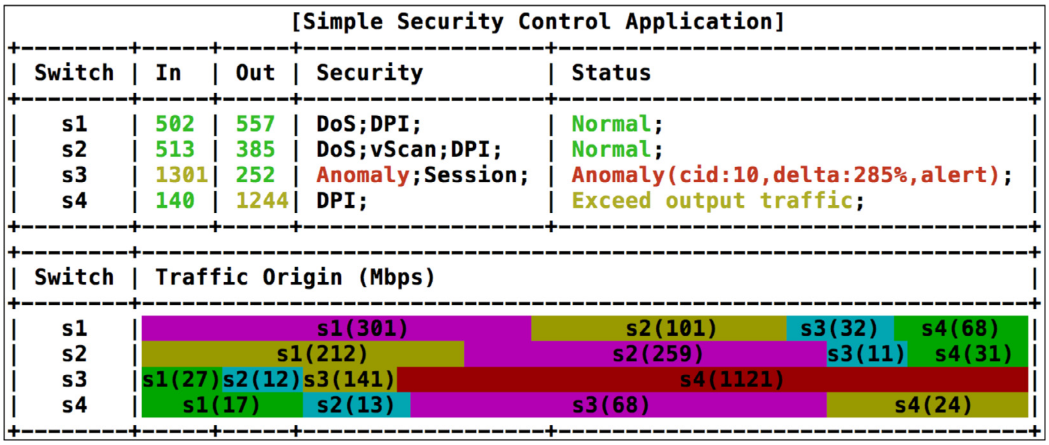
$$\sum_n \sum_i b_{n,i} \tag{1}$$

Subject to

$$\sum_i f^r_{n,i}(b_{n,i}) \le r^{MAX}_n \tag{2}$$

$$\sum_m b_{m,n} = \sum_o b_{n,o} \tag{3}$$

$$b_{sum\_of\_input} = b_{sum\_of\_output} \tag{4}$$



```
                 [Simple Security Control Application]
+--------+-----+-----+--------------------+----------------------------------+
| Switch | In  | Out | Security           | Status                           |
+--------+-----+-----+--------------------+----------------------------------+
|   s1   | 502 | 557 | DoS;DPI;           | Normal;                          |
|   s2   | 513 | 385 | DoS;vScan;DPI;     | Normal;                          |
|   s3   | 1301| 252 | Anomaly;Session;   | Anomaly(cid:10,delta:285%,alert);|
|   s4   | 140 | 1244| DPI;               | Exceed output traffic;           |
+--------+-----+-----+--------------------+----------------------------------+
+--------+-------------------------------------------------------------------+
| Switch | Traffic Origin (Mbps)                                             |
+--------+-------------------------------------------------------------------+
|   s1   |          s1(301)          |      s2(101)      | s3(32) | s4(68)   |
|   s2   |      s1(212)       |      s2(259)       | s3(11) |   s4(31)      |
|   s3   |s1(27)|s2(12)|s3(141)|                s4(1121)                    |
|   s4   |    s1(17)    | s2(13) |        s3(68)         |     s4(24)       |
+--------+-----+-----+--------------------+----------------------------------+
```

Fig. 14. A simple security control application with DPX.

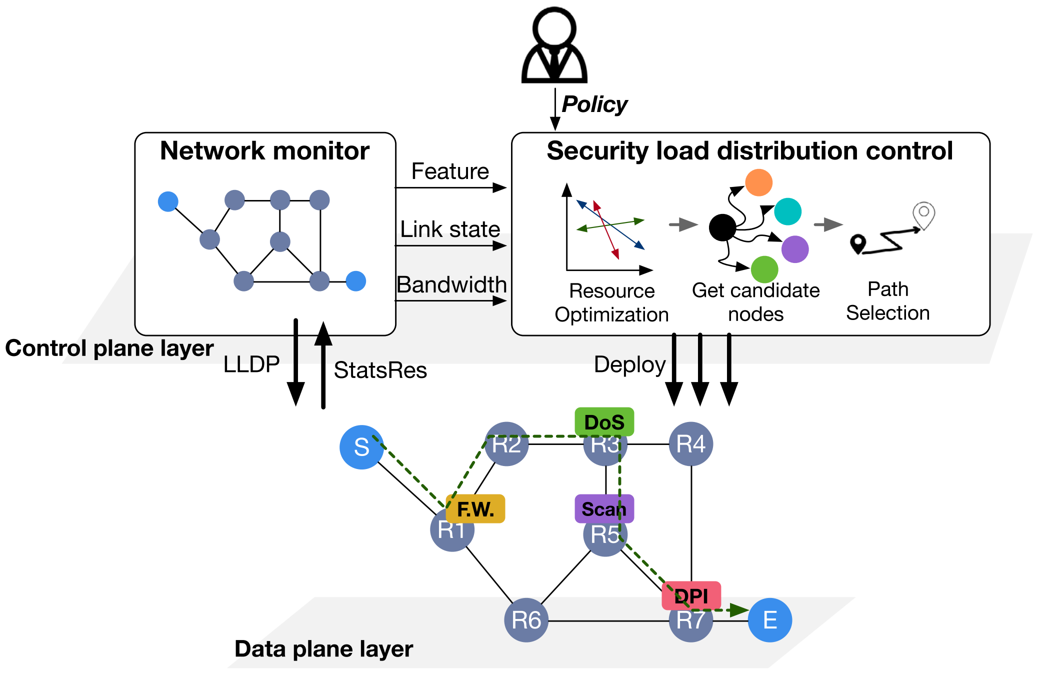


Fig. 15. Workflow of resource-aware action deployment.

The goal of the ILP is to derive maximum traffic amounts per security action running on each DPX (1). The traffic amount of security actions must not exceed resource capacities (2). The amount of incoming traffic at switch n must be the same with the amount of its outgoing traffic (3). Similarly, the total amount of incoming and outgoing traffic across a network must be the same (4).

After the execution of the ILP, the remaining traffic amounts for each security action are derived. DPX then chooses candidate nodes by referring to the administrator's security policy. After that, DPX derives a routing path whose link cost is low by combining candidate nodes.

6. Implementation

To verify the feasibility of the DPX design principle, we develop a prototype implementation of DPX in both software and hardware.

6.1. Software-based DPX switch

We implement the software-based DPX switch with 6K lines of C code based on the kernel datapath module of Open vSwitch (OVS)[2] v2.4.9 Open vSwitch (2022); Pfaff et al. (2015). Fig. 16 shows how DPX is integrated with the OVS modules. When a packet matched with the flow table comes to a switch, OVS invokes the `execute_actions` sequence with packet data (i.e., socket buffer) and an action key. The action key enumerates a list of actions that are executed for the matched packet, and each action in the list is sequentially called by the `execute_actions`. To enable DPX in this processing sequence, we have modified the `execute_actions` module of OVS to jump to the DPX entry point, the starting point for DPX security actions.

The DPX entry point will invoke a required security action by forwarding the socket buffer to the security action block. The

[2] Note that OVS is a software-based OpenFlow switch and it is widely employed in data-centers and enterprise networks today.

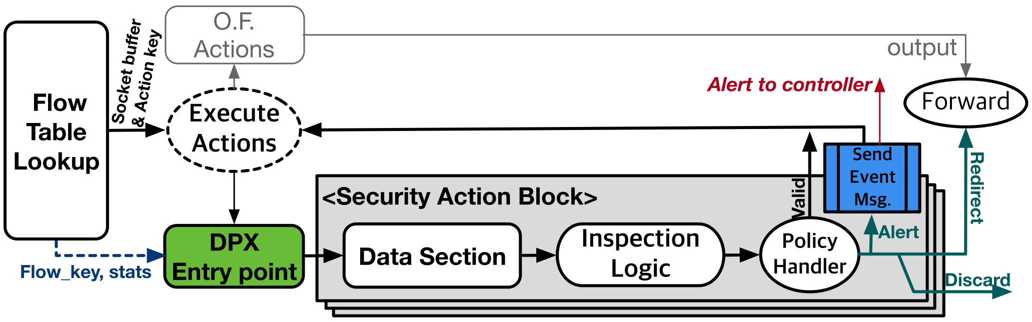


Fig. 16. The software-based DPX datapath.

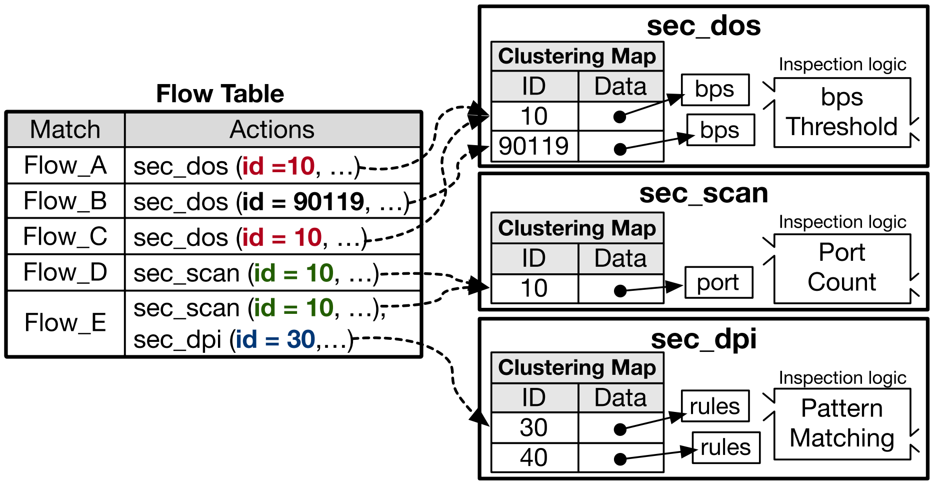


Fig. 17. Components and dataflow of the hardware-based DPX system architecture. The processing sequence includes the security action input selector, security action modules and a policy handler.

security action block is the main function code block performing three stages described in Section 4.2 (i.e., update the data section, perform the inspection logic, and impose the policy on detected traffic). Since a security action is composed of a modular function block, network operators can easily add a new security feature by registering a new block to the DPX entry point.

When executing a security action, the DPX switch forwards the socket buffer and the security instruction with its parameters to the DPX entry point. The entry point receives a `flow_key` with stats and invokes a proper security action block with them. Then, the security block updates the data section and performs the inspection logic. Once a security block processes a packet without detection, the DPX switch repeats the `execute_actions` sequence until all actions in the action key (including common OpenFlow actions) are processed.

The current version of the software-based DPX switch provides all features described in Table 2 and the total number of supported flows with an action cluster is limited only by the memory capacity of a host device. We have also extended the user-space of OVS to allow parsing incoming messages from DPX applications and delivering an event message from the kernel-space module to an SDN controller.

6.2. Hardware-based DPX switch

We implement the hardware-based DPX switch using the NetFPGA-SUME board,[3] an FPGA-based PCI Express board with four SFP+ 10 Gbps interfaces NetFPGA (2022). To enable DPX, we migrate the OpenFlow IP package of NetFPGA-10G board (NetFPGA GitHub Organization, 2012; Tatsuya Yabe, 2011) to our NetFPGA-SUME board and extend it to support DPX actions. We implement a security processing sequence on the hardware-based DPX, and it consists of three key modules: (i) a *security action input selector*, (ii) *security action modules*, and (iii) a *policy handler* (see Fig. 17).

The *security action input selector* is responsible for looking up security actions from the action key (Section 4.2) and forwarding their parameters to appropriate security action modules. All parameters are transferred through the wide data bus, and packet data is carried from the packet buffer to each security action module. A *security action module* is an independent entity that contains a data section through own memory space (We use a Block-RAM in this prototype.). Therefore, as shown in Fig. 17, all security action modules are executed in parallel at the same time. Due to this parallel processing, we separately place the security processing sequence in front of the OpenFlow action processor to avoid conflicts with OpenFlow packet modification actions (e.g., `set_nw_src`, `set_tp_src`). Finally, the *policy handler* executes the policies specified in security actions, i.e., `neglect`, `alert`, `discard`, `redirect` (Section 4.2). The current version of the hardware-based DPX switch provides the DoS detector with 1024 action clustering slots and the DPI action with four action clustering slots, each of which can store 1024 patterns, respectively.

Enabling communication with host To transfer messages generated from the hardware-based DPX switch to a controller or vice-versa, we also implement host software based on the reference implementation of NetFPGA-SUME. Fig. 18 illustrates the workflow for message transfer between the switch hardware and host software. They communicate through the device driver by reading

[3] Recently, NetFPGA-SUME has been widely used to prototype high-performance security devices, such as 100 Gbps IDS/IPS or network testing tools (Zilberman et al., 2014).

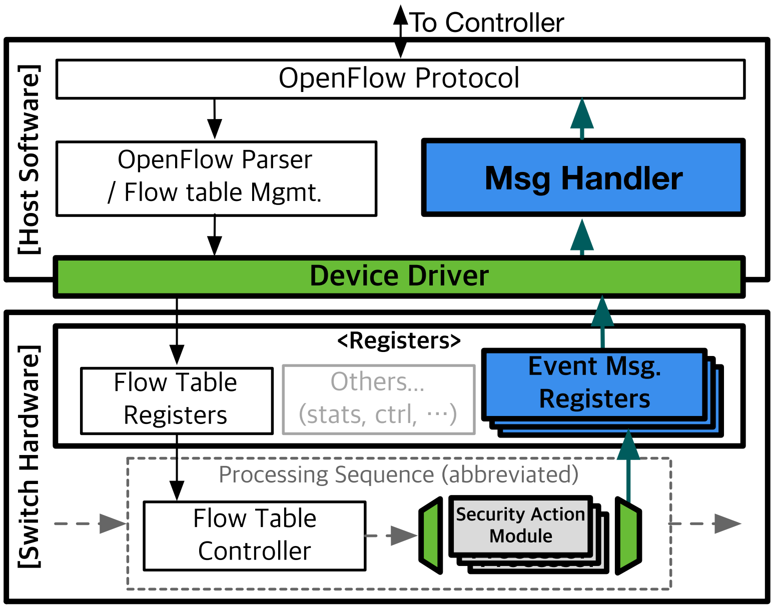


Fig. 18. Communication between the host software and switch hardware.

and writing registers on the NetFPGA-SUME. For example, when an `alert` message is sent to a controller, the message handler writes event information (e.g., event reason code, reference features, packet data, and cluster ID) to a corresponding register. The device driver delivers these register values to the DPX message handler in the host, and the message handler builds an OpenFlow message and sends it to the controller via the OpenFlow channel. In an opposite case, when an OpenFlow message arrives to deploy a new flow rule with security actions, the OpenFlow message parser writes them to the flow table registers to let the flow table controller perform security actions.

6.3. DPX controller

To assist network operators to program DPX applications easily, we design DPX APIs for the POX controller (POX, 2022), a Python-based SDN controller. We implement the DPX event handler class to receive DPX messages and a new Python module supporting DPX applications. We add around 500 lines of Python code to POX to enable all DPX related functions. Finally, we use the OpenFlow 1.0 vendor extension for communication between a DPX controller and DPX switches. Note that we choose POX and OpenFlow 1.0 due to their simplicity for rapid prototype development and feasibility evaluation. However, our design principles can be applied to recent OpenFlow versions and modern SDN controllers, such as ONOS (Berde et al., 2014), OpenDaylight (Medved et al., 2014), or Floodlight Project Floodlight (0000).

7. Performance evaluation

This section presents results from our system performance evaluation (specifically, throughput, latency, and computational overhead) as well as results that illustrate the benefit of DPX's flow-table simplification.

7.1. Test environment

The test environment (Fig. 19) consists of two hosts (i.e., `h1` and `h2`) and an NFV host with a datapath device that operates the DPX switch and the DPX controller. All host machines run Ubuntu 14.04 and have an Intel Xeon E5-2630@2.9 GHz processor, 64 GB

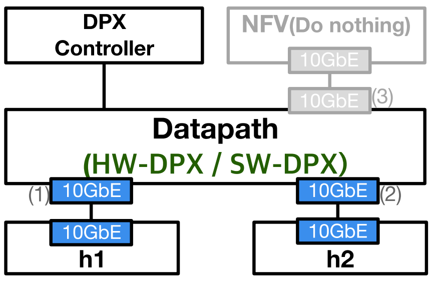


Fig. 19. The testbed for performance evaluation.

of RAM, and Intel X520-DA2 10GbE NICs. The datapath device uses the NetFPGA-SUME board for running the hardware-based DPX switch, or Open vSwitch v2.4.90 for running the software-based DPX switch. Although DPX is likely to be deployed in multi-switch environments, we focus on a single switch benchmark because we assume that a bottleneck within a switch determines the overall throughput.

To evaluate DPX, we configure the DPX switch to forward all incoming packets from `h1` to `h2` after executing DPX actions (e.g., `DoS`, `DPI`[4], `Chain`). Then, we compare DPX to two different cases: First, we measure the performance of the native software and hardware switches where no security solutions are deployed as baseline (i.e., `simple`). For this, we configure them to forward packets from `h1` to `h2` without further processing. Second, we measure the performance when packets traverse a single NFV node before arriving at `h2` (i.e., `NFV`). Identical way to the experiment conducted in Section 2.1, the NFV node does nothing and immediately returns packets to the DPX switch. Note that we do *not* aim to benchmark an individual DPX security action or verify their functionality. Instead, we aim to assess (i) the performance overhead of DPX over no security solution case and (ii) the performance improvement over the NFV-based security solution.

For performance metrics, we measure (i) end-to-end throughput with `Intel DPDK-Pktgen` (Intel, 2022a) by generat-

[4] Note that we mark the *number of rules* at the suffix of `DPI` (e.g., `DPI100`).

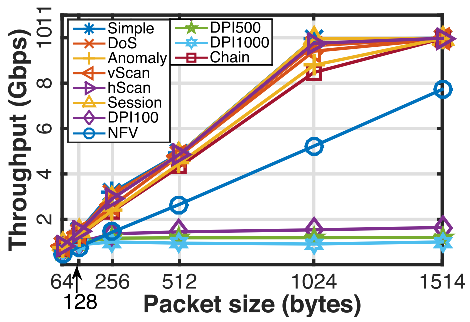


Fig. 20. Throughput of the software-based DPX.

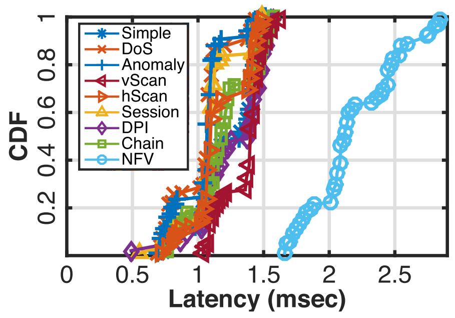


Fig. 21. Latency of the software-based DPX.

ing various sizes of packet bursts and (ii) latency with `nping` (Nping, 2022) through the RTT of TCP packets containing random 256-byte payloads.

7.2. Performance of software-based DPX

Throughput Fig. 20 illustrates the measured throughput from the software-based DPX switch. We can see that most of the DPX actions (except for `DPI`) incur small overheads compared to the native software switch (i.e., `simple`). Specifically, they achieve at least 90% throughput of the baseline in all packet sizes. In the case of deploying a service chain (i.e., `Chain`) that comprises all the DPX actions except DPI, only a minor performance degradation is observed, similar to the overhead of `Anomaly`. Our analysis concludes that the degradation is not the overhead by the chaining itself but mainly the bottleneck by the worst-performing security action in the chain.

On the other hand, the `DPI` action only achieves 1 Gbps throughput when using 100 rules, and it is even degraded with the growth of rules (i.e., `DPI500` and `DPI1000`). We discover that this limitation is attributed to the overhead of pattern matching in software. For example, popular IDS software, such as Snort (2022) and Suricata (2022), also achieve approximately 0.8–1.2 Gpbs in our experiments. It is difficult to directly compare the DPX `DPI` action with Snort and Suricata because of functional differences. However, this result suggests that the `DPI` action could be utilized at edge switches before forwarding to NFV nodes for more deep packet inspection.

Note that the NFV-based approach only achieves the throughput of 7.726 Gbps in the best case (i.e., 1514-byte). At first glance, one could argue that the throughput of the NFV-based approach must be similar to the baseline because traffic steering is based on simple forwarding. However, we observe a performance degradation factor besides the bottleneck on the NFV host. It mainly stems from the fact that the software switch (e.g., OVS) processes two packet streams concurrently. More specifically, when the host `h1` sends a packet stream to the software switch, the NFV host also sends a packet stream to the switch at the same time. Thus, the bandwidth capacity of the software switch can be exceeded easily, degrading the overall throughput of the network.

Latency Fig. 21 illustrates CDF of the latency measured from the software-based DPX. The latency of DPX actions is close to that of the native software switch, including the DPI action. Further, there is no significant overhead while constructing a service chain. For example, 99% of packets are processed in less than 1.5 ms, similar to the baseline. In contrast, the average latency of using the NFV node is 2.180 ms while it is 1.213 ms in the native software switch. Thus, we could see that the NFV node incurs about twice the baseline latency.

7.3. Performance of hardware-based DPX

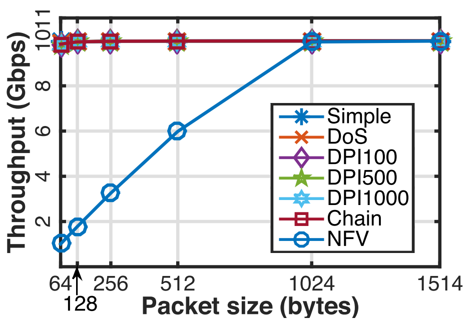
Throughput Fig. 22(a) illustrates the throughput measured in the hardware-based DPX switch. We can see that all DPX security actions (i.e., `DoS`, `DPI` with 100, 500 and 1000 rules) achieve throughput close to 10 Gbps. On closer inspection (Fig. 22(b)), while DPX security actions incur throughput degradation ($<$ 1% reduction) in the worst case (64-byte packet size), the line-rate performance is achieved as the packet size increases. In addition, when configuring a service chain with DoS and DPI (i.e., `Chain`), we find that there is no observable overhead because of the parallel processing provided by hardware-based DPX. In contrast, the NFV-based approach degrades the throughput significantly before the 1024-byte packet size. In particular, it achieves only 1 Gbps of throughput at the 64-byte packet size. This degradation is mainly caused by the bottleneck on the NFV host and processing overhead of the incoming and outgoing packet stream. Whereas this throughput degradation can be moderated through improvements of a host machine, it is difficult to eradicate the bottleneck, given that many VMs are service-chained in real deployments.

Latency Fig. 23 illustrates the latency measured in the hardware-based DPX switch. In most cases, the latency of when DPX actions are deployed is similar to the one measured the case where no security solutions are deployed (i.e., `simple`). It is shown that 99% of packets are processed in less than 0.65 ms. Even in the case of the service chains, there is no meaningful overhead in latency. This result is remarkable when we compare DPX actions with the NFV host. Although the NFV host directly returns traffic without any additional processing, the latency is a factor of two or more times higher than DPX actions.

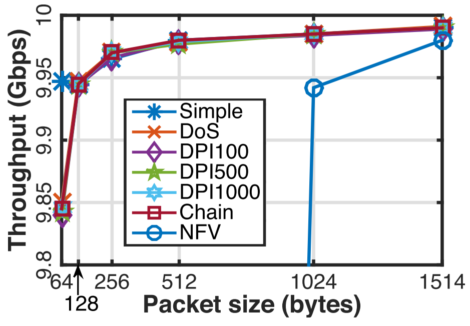
Computational overhead We also measure the computational overhead of DPX actions, and it was negligible (1-2%) in comparison to the packet switching overhead.

7.4. Performance of DPX network actions

To validate possibilities for further extension besides the security features, we evaluate DPX network actions, i.e., `NAT`, `Load-balancer`, and `ARP proxy` (see Table 2). The experiments were performed on the software-based DPX. Fig. 24 shows the throughput achieved for different message sizes on `NAT` and `Load-balancer` actions. DPX network actions exhibit lower overhead than security actions, nearly close to that of the native software switch. This is because most network actions are relatively lightweight than security actions; they only need to perform a forwarding decision and a packet header manipulation. Fig. 25 illustrates the CDF of the latency measured when DPX network actions are deployed. In most cases, the latency of network actions approach the software switch. For example, 99% of packets including the native software switch are processed less than 1.5 msec.



(a) Full-size graph.



(b) Magnified graph.

Fig. 22. Throughput of the hardware-based DPX.

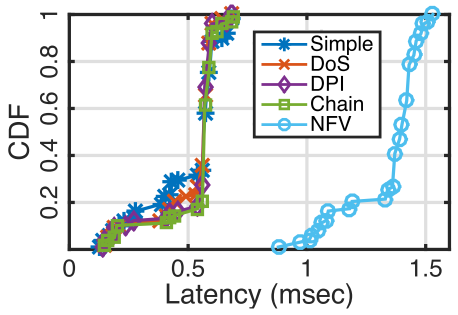


Fig. 23. Latency of the hardware-based DPX.

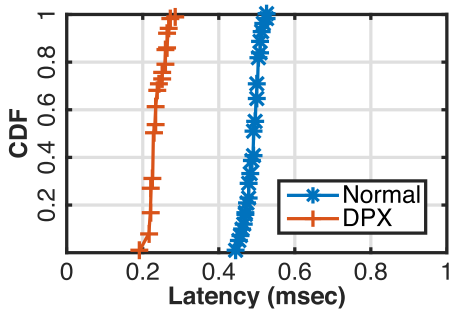


Fig. 26. Latency of ARP response.

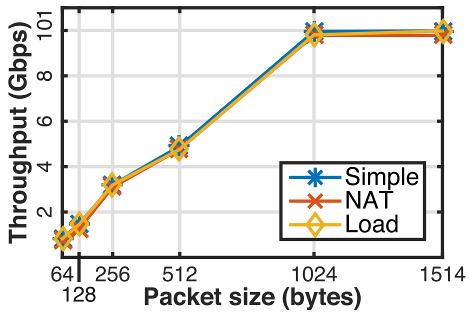


Fig. 24. Throughput of DPX network actions.

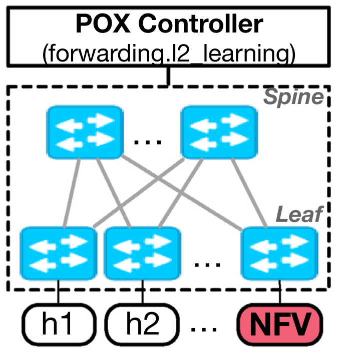


Fig. 27. Leaf-spine topology.

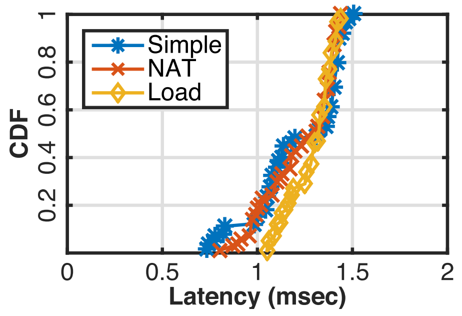


Fig. 25. Latency of DPX network actions.

Therefore, the overhead can be considered as jitter and possibly ignored.

We also evaluate the response time of the `ARP proxy` action using `arping` (ARping, 2022) and measured the RTT time of ARP packets using `tcpdump`. We compare ARP response times of the action to a normal ARP behavior between `h1` and `h2`. Fig. 26 illustrates the measured ARP response time. The `ARP proxy` action shows shorter response time about two or more time than the normal ARP behavior. It is a natural result given that the `ARP proxy` action eliminates the need for delivering an ARP message to an end host. Hence, the response time decreases as much as the reduced traveling path.

7.5. Effectiveness of rule simplification

We evaluate the effectiveness of DPX in simplifying flow rules. Because the size of switch flow tables varies depending on various factors, such as configurations, network policies, and traffic, it is difficult to make universal claims. Hence, we assume a specific use case and emulate it using `Mininet` (Lantz et al., 2010) and the POX controller.

We reproduce a leaf-spine topology that is a two-layer datacenter network architecture, and connected end hosts to each leaf switch as depicted in Fig. 27. One of the connected hosts is used as an NFV host to operate network services. Then, we count the number of required flow rules when all hosts can communicate with each other (i.e., using *ping-all* test without packet loss) including a path to visit an NFV service chain, while increasing the number of hosts. The flow rules are installed by the *forwarding.l2_learning* application on the POX controller.

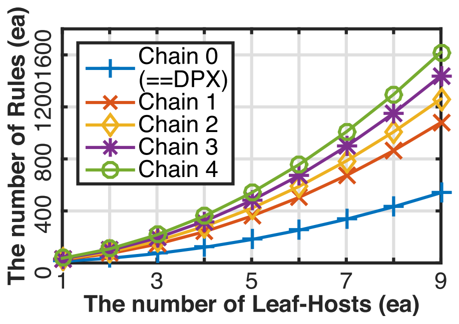


Fig. 28. The count of required rules.

As shown in Fig. 28, the number of required flow rules for the entire network exponentially increases following the number of hosts and the length of the service chain to drive traffic to the NFV host and a service chain. When the number of leaf hosts is 10 and the length of the service chain is four, the network needs 1620 rules for the communications between all hosts. On the other hand, DPX can provide a service chain without any detouring of traffic. Thus, the network with DPX only requires minimal flow rules that lead traffic to their destination directly, and it is equivalent to when the length of the service chain is zero. Therefore, the number of required flow rules is significantly reduced regardless of the length of the service chain. Specifically, even if the number of leaf hosts is 10 and the length of service chain is four, the network only needs 540 rules to enable communications between all hosts.

8. Discussion

In this section, we discuss related issues around DPX.

Performance benefit Overall, we could see that NFV service chains induce a significant additional overhead while individual latency varies. Given that NFV service chains in real networks are likely to be complex, the degradation will be more significant than the experiments. In this respect, the DPX approach that provides security services from switches without traffic steering is beneficial.

Management benefit We show that DPX can reduce the number of flow rules and the action clustering allows complex security policies to be expressed in a simplified flow table. Today, many hosts exist on a network thanks to virtualization technology. Optimizing traffic engineering in terms of performance and security while considering NFV node placement requires complicated rules. Thus, simplifying security policies with such techniques will help relieve the administrator's management complexities.

Security analysis One could argue that an attacker may attempt to leak confidential data (e.g., network policies) by performing fingerprinting attacks against SDN switches (Shin and Gu, 2013; Sonchack et al., 2016b). In contrast to OpenFlow, the execution result of DPX security actions is irrelevant to match fields, exposing no prominent patterns to attackers. Also, as DPX is fully compatible with OpenFlow, all control packets including DPX secuirty actions can be encrypted with TLS/SSL. Thus, even though an attacker captures packets, it is impossible to obtain valuable information.

Extensibility At first glance, it is difficult to add a new function to DPX since its execution environment is in a switch where developing programs are restricted than usual. In the case of software-based DPX, it is designed as modular components; thus, DPX can be simply extended by registering a new action block to the internal interface (as mentioned in Section 6). In the case of hardware-based DPX built on FPGA, many companies, such as Microsoft, have employed FPGAs in their data centers (Firestone et al., 2018). Thus, we believe that practitioners who have sufficient knowledge of FPGAs could implement additional components of the hardware-based DPX without obstacles.

Backward compatibility Network operators can still use existing OpenFlow commands and rules because DPX is designed to support the legacy OpenFlow protocol as well. One possible issue is when OpenFlow messages containing DPX actions are delivered to a non-DPX switch. The switch could be put into an unpredictable state because the DPX actions will not be parsed correctly. To avoid this, the DPX controller removes DPX actions from the message and runs a corresponding SDN application (e.g., a DDoS detector application instead of the `sec_dos` action). This way, it is possible to keep the backward compatibility while ensuring the functional requirement at the cost of performance loss, i.e., control-plane bottleneck.

9. Conclusion and future work

In this paper, we present the design and implementation of the new data plane architecture called DPX, a switch-native solution for efficient yet high-performance security functions. We show that DPX simplifies composition of service chains and enables graceful integration of security functions without associated detouring overheads. In addition, we propose action clustering, eliminating the redundant packet processing within a switch by integrating multiple actions into a single one. Our evaluation demonstrates that DPX achieves significantly improved throughput and latency than the NFV deployment cases. The several use-cases show that DPX can successfully prevent all network attacks while compressing the number of required flow rules.

We sketch a possible future work associated with the data plane extension. Recently, programmable data planes have garnered significant attention. Whereas they offer benefits over SDN in terms of flexibility, several essential security functions, such as deep packet inspection (DPI), are still not supported. Thus, extending the programmable switches to support advanced security functions would be an interesting research topic.

Data availability

Extended data plane architecture for in-network security services in software-defined networks.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Jinwoo Kim: Conceptualization, Methodology, Validation, Writing – review & editing. **Yeonkeun Kim:** Conceptualization, Methodology, Writing – original draft. **Vinod Yegneswaran:** Conceptualization, Validation, Writing – review & editing. **Phillip Porras:** Conceptualization, Validation, Writing – review & editing. **Seungwon Shin:** Conceptualization, Formal analysis, Project administration, Supervision, Funding acquisition, Writing – original draft, Writing – review & editing. **Taejune Park:** Conceptualization, Software, Formal analysis, Project administration, Supervision, Funding acquisition, Writing – original draft, Writing – review & editing.

Acknowledgment

This work was supported in part by National Research Foundation of Korea (No. 2022R1C1C1006967).

References

Anderson, J.W., Braud, R., Kapoor, R., Porter, G., Vahdat, A., 2012. xOMB: extensible open middleboxes with commodity servers. In: Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems, pp. 49–60.

Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J.A., Invernizzi, L., Kallitsis, M., et al., 2017. Understanding the mirai botnet. In: Proceedings of the 26th USENIX Security Symposium (USENIX Security '18), pp. 1093–1110.

ARping, 2022. Ping destination on device interface by ARP packets. http://www.habets.pp.se/synscan/programs.php?prog=arping.

Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., Koide, T., Lantz, B., O'Connor, B., Radoslavov, P., Snow, W., et al., 2014. ONOS: towards an open, distributed SDN OS. In: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking. ACM, pp. 1–6.

BlueCat Networks, 2022. Making the Case for SDN: A Real-World Example. https://bluecatnetworks.com/blog/making-case-sdn-real-world-example/.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al., 2014. P4: programming protocol-independent packet processors. ACM SIGCOMM Comput. Commun. Rev. 44 (3), 87–95.

Bremler-Barr, A., Harchol, Y., Hay, D., 2016. Openbox: a software-defined framework for developing, deploying, and managing network functions. In: Proceedings of the 2016 ACM SIGCOMM Conference, pp. 511–524.

Casado, M., Freedman, M.J., Pettit, J., Luo, J., McKeown, N., Shenker, S., 2007. Ethane: taking control of the enterprise. ACM SIGCOMM Comput. Commun. Rev. 37 (4), 1–12.

Fayaz, S.K., Tobioka, Y., Sekar, V., Bailey, M., 2015. Bohatei: flexible and Elastic DDoS Defense. In: Proceedings of the 24th USENIX Security Symposium (USENIX Security 15). USENIX Association, pp. 817–832.

Fayazbakhsh, S.K., Chiang, L., Sekar, V., Yu, M., Mogul, J.C., 2014. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In: Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14), pp. 543–546.

Ferguson, A.D., Gribble, S., Hong, C.-Y., Killian, C., Mohsin, W., Muehe, H., Ong, J., Poutievski, L., Singh, A., Vicisano, L., et al., 2021. Orion: google's software-defined networking control plane. In: Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18), pp. 83–98.

Firestone, D., Putnam, A., Mundkur, S., Chiou, D., Dabagh, A., Andrewartha, M., Angepat, H., Bhanu, V., Caulfield, A., Chung, E., et al., 2018. Azure accelerated networking: smartnics in the public cloud. 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), Renton, WA.

Gember-Jacobson, A., Viswanathan, R., Prakash, C., Grandl, R., Khalid, J., Das, S., Akella, A., 2014. OpenNF: enabling innovation in network function control. ACM SIGCOMM Comput. Commun. Rev. 44 (4), 163–174.

Greenberg, A., Hjalmtysson, G., Maltz, D.A., Myers, A., Rexford, J., Xie, G., Yan, H., Zhan, J., Zhang, H., 2005. A clean slate 4D approach to network control and management. ACM SIGCOMM Comput. Commun. Rev. 35 (5), 41–54.

Gupta, A., Habib, M.F., Mandal, U., Chowdhury, P., Tornatore, M., Mukherjee, B., 2018. On service-chaining strategies using virtual network functions in operator networks. Comput. Netw. 133, 1–16.

Honda, M., Huici, F., Lettieri, G., Rizzo, L., 2015. mSwitch: a highly-scalable, modular software switch. In: Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research. ACM, New York, NY, USA, pp. 1:1–1:13.

Hong, C.-Y., Mandal, S., Al-Fares, M., Zhu, M., Alimi, R., Bhagat, C., Jain, S., Kaimal, J., Liang, S., Mendelev, K., et al., 2018. B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined WAN. In: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18), pp. 74–87.

hping3, 2022. A network tool able to send custom TCP/IP packets and to display target replies. http://www.hping.org/hping3.html.

Hwang, J., Ramakrishnan, K.K., Wood, T., 2014. NetVM: high performance and flexible networking using virtualization on commodity platforms. In: Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). USENIX Association, Seattle, WA, pp. 445–458.

Intel, 2022. Intel DPDK: Data Plane Development Kit. http://dpdk.org.

Intel, 2022. Intel Tofino Series Programmable Ethernet Switch ASIC. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

Kampanakis, P., Perros, H., Beyene, T., 2014. SDN-based solutions for moving target defense network protection. In: Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014. IEEE, pp. 1–6.

Kang, M.S., Gligor, V.D., Sekar, V., et al., 2016. SPIFFY: inducing cost-detectability tradeoffs for persistent link-flooding attacks. In: Proceedings of the Network and Distributed Systems Security Symposium.

Kim, H., Feamster, N., 2013. Improving network management with software defined networking. IEEE Commun. Mag. 51 (2), 114–119.

Lantz, B., Heller, B., McKeown, N., 2010. A network in a laptop: rapid prototyping for software-defined networks. In: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotSDN '10). ACM, p. 19.

Lee, S., Kim, J., Shin, S., Porras, P., Yegneswaran, V., 2017. Athena: a framework for scalable anomaly detection in software-defined networks. In: 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '17). IEEE, pp. 249–260.

Li, G., Zhang, M., Guo, C., Bao, H., Xu, M., Hu, H., Li, F., 2022. IMap: fast and scalable in-network scanning with programmable switches. In: Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI '22), pp. 667–681.

Liu, G., Guo, S., Li, P., Liu, L., 2020. Conmidbox: consolidated middleboxes selection and routing in SDN/NFV-enabled networks. In: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, pp. 946–955.

Liu, Z., Namkung, H., Nikolaidis, G., Lee, J., Kim, C., Jin, X., Braverman, V., Yu, M., Sekar, V., 2021. Jaqen: a high-performance switch-native approach for detecting and mitigating volumetric DDoS attacks with programmable switches. In: 30th USENIX Security Symposium (USENIX Security 21), pp. 3829–3846.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J., 2008. OpenFlow: enabling innovation in campus networks. In: Proceedings of ACM SIGCOMM Computer Communication Review.

Medved, J., Varga, R., Tkacik, A., Gray, K., 2014. Opendaylight: towards a model–driven SDN controller architecture. In: World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on a. IEEE, pp. 1–6.

Mekky, H., Hao, F., Mukherjee, S., Lakshman, T., Zhang, Z.-L., 2017. Network function virtualization enablement within SDNdata plane. In: IEEE INFOCOM, pp. 1–9.

Metasploit, 2022. Penetration Testing Software. https://www.metasploit.com/.

Nam, J., Seo, J., Shin, S., 2018. Probius: automated approach for VNF and service chain analysis in software-defined NFV. In: Proceedings of the Symposium on SDN Research (SOSR '18).

NetFPGA. NetFPGA-SUME board. https://netfpga.org/site/#/systems/1netfpga-sume/details/.

NetFPGA GitHub Organization, 2012. Netfpga 10G openflow switch. https://github.com/NetFPGA/NetFPGA-public/wiki/NetFPGA-10G-OpenFlow-Switch.

nmap, 2022. Network Mapper - Security Scanner. https://nmap.org/.

Nping, 2022. An Open source network packet generation. https://nmap.org/nping/.

Open vSwitch. An Open Virtual Switch. http://openvswitch.org/.

OpenFlow, 2022. Open Networking Foundation (ONF). https://www.opennetworking.org/sdn-resources/openflow.

Park, T., Kim, Y., Park, J., Suh, H., Hong, B., Shin, S., 2016. QoSE: quality of security a network security framework with distributed NFV. In: Communications (ICC), 2016 IEEE International Conference on. IEEE, pp. 1–6.

Park, T., Nam, J., Na, S.H., Chung, J., Shin, S., 2021. Reinhardt: real-time reconfigurable hardware architecture for regular expression matching in DPI. In: Annual Computer Security Applications Conference (ACSAC '21), pp. 620–633.

Park, T., Shin, S., 2021. Mobius: packet re-processing hardware architecture for rich policy handling on a network processor. J. Netw. Syst. Manag. 29 (1), 1–26.

Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., Amidon, K., Casado, M., 2015. The design and implementation of open vswitch. In: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15). USENIX Association, Oakland, CA, pp. 117–130. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff.

Pica. PicOS Support for OpenFlow 1.3. https://docs.pica8.com/display/PICOS2111cg/PicOS+Support+for+OpenFlow+1.3.

POX, 2022. Python Network Controller. http://www.noxrepo.org/pox/about-pox/.

Project Floodlight,. Open Source Network Operating System. http://www.projectfloodlight.org/floodlight/.

Qazi, Z.A., Tu, C.-C., Chiang, L., Miao, R., Sekar, V., Yu, M., 2013. Simple-fying middlebox policy enforcement using SDN. In: Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM. ACM, New York, NY, USA, pp. 27–38. doi:10.1145/2486001.2486022.

Sekar, V., Egi, N., Ratnasamy, S., Reiter, M.K., Shi, G., 2012. Design and implementation of a consolidated middlebox architecture. In: Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12). USENIX, San Jose, CA, pp. 323–336. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/sekar.

Shin, S., Gu, G., 2012. Cloudwatcher: network security monitoring using openflow in dynamic cloud networks (or: how to provide security monitoring as a service in clouds?). In: Network Protocols (ICNP), 2012 20th IEEE International Conference on. IEEE, pp. 1–6.

Shin, S., Gu, G., 2013. Attacking software-defined networks: a first feasibility study. In: Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, pp. 165–166.

Shin, S., Xu, L., Hong, S., Gu, G., 2016. Enhancing network security through software defined networking (SDN). In: Proceedings of the 25th International Conference on Computer Communication and Networks (ICCCN). IEEE, pp. 1–9.

Shin, S., Yegneswaran, V., Porras, P., Gu, G., 2013. Avant-guard: scalable and vigilant switch flow management in software-defined networks. In: Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS 2013).

Snort. Network Intrusion Detection System. https://www.snort.org/.

Sonchack, J., Aviv, A. J., Keller, E., Smith, J. M., 2016a. Enabling practical software-defined networking security applications with ofx.

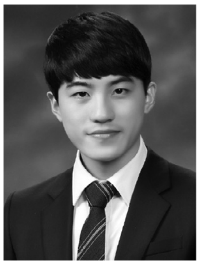
Sonchack, J., Dubey, A., Aviv, A.J., Smith, J.M., Keller, E., 2016. Timing-based reconnaissance and defense in software-defined networks. In: Proceedings of the 32nd Annual Conference on Computer Security Applications, pp. 89–100.

Suricata, 2022. An open source-based intrusion detection system (IDS). https://suricata-ids.org/.

Tatsuya Yabe, 2011. Openflow implementation on NetFPGA-10G design document.

Xing, J., Wu, W., Chen, A., 2021. Ripple: a programmable, decentralized link-flooding defense against adaptive adversaries. In: Proceedings of the 30th USENIX Security Symposium (USENIX Security '21), pp. 3865–3881.

Yang, X., Han, B., Sun, Z., Huang, J., 2017. SDN-based DDoS attack detection with cross-plane collaboration and lightweight flow monitoring. In: GLOBECOM 2017-2017 IEEE Global Communications Conference. IEEE, pp. 1–6.
Yoon, C., Park, T., Lee, S., Kang, H., Shin, S., Zhang, Z., 2015. Enabling security functions with SDN: a feasibility study. Comput. Netw. 85, 19–35.
Yu, R., Xue, G., Kilari, V.T., Zhang, X., 2015. Network function virtualization in the multi-tenant cloud. IEEE Netw. 29 (3), 42–47.
Yu, T., Fayaz, S.K., Collins, M.P., Sekar, V., Seshan, S., 2017. Psi: precise security instrumentation for enterprise networks. NDSS.
Zhang, M., Li, G., Wang, S., Liu, C., Chen, A., Hu, H., Gu, G., Li, Q., Xu, M., Wu, J., 2020. Poseidon: mitigating volumetric DDoS attacks with programmable switches. In: Proceedings of the 27th Network and Distributed System Security Symposium (NDSS '20).
Zilberman, N., Audzevich, Y., Covington, G.A., Moore, A.W., 2014. NetFPGA SUME: toward 100 Gbps as research commodity. IEEE Micro 34 (5), 32–41.

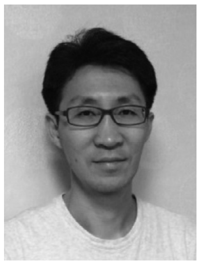


Jinwoo Kim is an assistant professor in the School of Software at Kwangwoon University, Seoul, South Korea. He received his Ph.D. degree in School of Electrical Engineering and his M.S. degree in Graduate School of Information Security from KAIST, and his B.S. degree from Chungnam National University in Computer Science and Engineering. His research topicminaly focuses on investigating security issues with software defined networks and cloud computing systems.

YeonKeun Kim is a Ph.D. student in the Graduate School of Information Security at KAIST. He received his B.S. degree in Computer Science Engineering at Ulsan National Institute of Science and Technology (UNIST) in Korea. He received his M.S. degree in Information Security from KAIST. His research interests include network security issues of IoT and embedding systems.

Vinod Yegneswaran received his A.B. degree from the University of California, Berkeley, CA, USA, in 2000, and his Ph.D. degree from the University of Wisconsin, Madison, WI, USA, in 2006, both in Computer Science. He is a Senior Computer Scientist with SRI International, Menlo Park, CA, USA, pursuing advanced research in network and systems security. His current research interests include SDN security, malware analysis and anti-censorship technologies. Dr. Yegneswaran has served on several NSF panels and program committees of security and networking conferences, including the IEEE Security and Privacy Symposium.

Phillip Porras received his M.S. degree in Computer Science from the University of California, Santa Barbara, CA, USA, in 1992. He is an SRI Fellow and a Program Director of the Internet Security Group in SRI's Computer Science Laboratory, Menlo Park, CA, USA. He has participated on numerous program committees and editorial boards, and participates on multiple commercial company technical advisory boards. He continues to publish and conduct technology development on numerous topics including intrusion detection and alarm correlation, privacy, malware analytics, active and software defined networks, and wireless security.



Seungwon Shin is an associate professor in the School of Electrical Engineering at KAIST. He received his Ph.D. degree in Computer Engineering from the Electrical and Computer Engineering Department, Texas A&M University, and his M.S. degree and B.S. degree from KAIST, both in Electrical and Computer Engineering. He is currently a corporate vice president at Samsung Electronics, leading the security team in the IT & Mobile Communications Devision. His research interests span the areas of Software-defined networking security, IoT security, Botnet analysis/detection, DarkWeb analysis and cyber threat intelligence.

Taejune Park is an assistant professor at the Department of Artificial Intelligence Convergence, Chonnam National University, South Korea. He received B.S. in Computer Engineering at Korea Maritime and Ocean University, South Korea, and M.S. and Ph.D. in information security at KAIST, South Korea. His research interests focus on network and IoT security and reliable/low-latency communications