DPX: Data-Plane eXtensions for SDN Security Service Instantiation

Taejune Park[1], Yeonkeun Kim[1], Vinod Yegneswaran[2], Phillip Porras[2], Zhaoyan Xu[3], KyoungSoo Park[1], and Seungwon Shin[1]

[1] KAIST, Daejeon, Republic of Korea
taejune.park@kaist.ac.kr
[2] SRI International, CA, USA
[3] Palo Alto Networks, CA, USA

Abstract. SDN-based 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, SDN-based NFV services often suffer from poor performance and require complex configurations due to the fact that 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 that natively supports security services as a set of abstract security actions that are then translated to OpenFlow rule sets. The DPX action model reduces redundant processing caused by frequent packet parsing and provides administrators 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. We present an implementation of DPX in hardware using NetFPGA-SUME and in software using Open vSwitch. We evaluated 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. We find that DPX imposes minimal overheads in terms of latency (≈0.65 ms in hardware and ≈1.2 ms in software on average) and throughput (≈1% of simple forwarding in hardware and ≈10% in software for non-DPI security services). This translates to an improvement of 30% over traditional NFV services on the software implementation and 40% in hardware.

1 Introduction

Modern enterprise and cloud network service management increasingly relies on techniques such as software-defined networking (SDN) and network function virtualization (NFV). One driver for this change is the flexibility that is afforded by the transition from specialized hardware devices to virtualized software images that run on commodity computing hardware. This “softwarization” trend is welcomed by network administrators since it facilitates elastic scaling and dynamic resource provisioning.

However, the implementation and deployment of these techniques also raise several practical deployment challenges. First, to fully leverage the benefits of NFV and SDN, the network needs to carefully incorporate a network orchestration strategy [11], such that an NFV integration does not overly complicate the network management environment (i.e., a *management challenge*). For example, to operate multiple network functions efficiently, a network administrator may produce an orchestration strategy that results in a diverse set of associated network flow rules. Optimizing the resulting flow orchestration is important. As NFV services are instantiated as software instances, separate from network devices, they have the potential to degrade network service performance when compared with legacy hardware-based solutions (i.e., a *performance challenge*). We make the case that no contemporary system comprehensively addresses both of the aforementioned challenges.

To address the management challenge, we are informed by prior projects on efficient orchestration of virtualized network functions and middleboxes (e.g., CoMb [28] and Bohatei [3]). However, these efforts primarily focus on coordinating network services (i.e., at the control plane) and do not address the underlying complexity of managing network flow rules (i.e., at the data plane). While recent efforts such as ClickOS [13] and NetVM [9] attempt to improve NFV performance by reducing management overhead using sophisticated I/O handling techniques, NFV systems still suffer from structural performance overhead that stems from "traffic detouring".

This paper explores and evaluates one approach to streamlining NFV flow processing through the extension of native services directly within the SDN data plane. Currently, most SDN data-plane device (i.e., SDN switch) implementations merely support basic packet-handling logic (e.g., forward, drop, and modify headers). However, as SDN switches also include various processing elements (e.g., storing packets and parsing headers), these may be leveraged for embedding additional security-service and management logic. Thus, we pose the following research questions:

- Can an SDN data plane provide more advanced features, such as payload inspection and malicious traffic filters?
- Could these features be exploited to significantly reduce the performance overhead and the complexity of NFV orchestration?

For example, consider a switch that locally filters disallowed packets, which provides an operator the ability to short-circuit the redundant forwarding of traffic to a firewall immediately from the network. Recent advances in high-performance switching [7, 20], suggests that this could potentially be a viable and attractive solution. Inspired by the potential benefits that can arise form the encapsulation of light-weight security primitives into the SDN data-plane, we present a design and prototype implementation of DPX. DPX is designed as an OpenFlow data plane [21] extension that natively supports security services as a set of OpenFlow actions. DPX can not only reduce redundant duplicated processing caused by frequent packet parsing to reduce latencies, but also represent them as OpenFlow rules, allowing administrators to easily configure network security services with simplified flow tables.

An important challenge to address in making this leap is the ability to express security policies over aggregated flow sets. Such flow set expressions are necessary because of the prohibitive expense of representing security rules using per-flow rules (i.e., the *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.

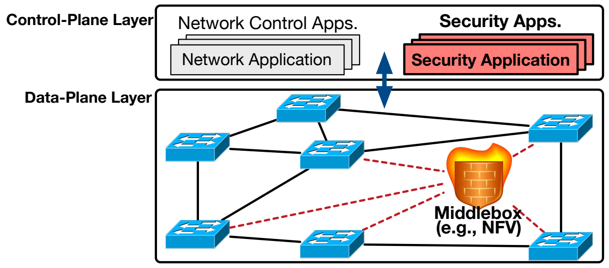


Fig. 1. Two deployment strategies for security services in SDNs (control-plane applications and in-network middleboxes)

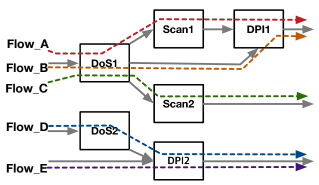
DPX is designed in a manner that attempts to preserve the original philosophy of SDN (i.e., simple data plane) to the extent possible by constructing its extensions as modular SDN data plane components. As we noted, newly added security actions are realized by OpenFlow actions, and those new security 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 seeks to detect DDoS attacks, the DPX SDN data plane can be extended in a component-wise manner, by simply adding a DoS detection module.

We have implemented a DPX prototype with six security services in software and two in hardware, using Open vSwitch [20] and NetFPGA-SUME [17] respectively. DPX achieves a throughput of 9.9 Gbps, with 0.65 ms of added latency, producing a performance profile that is comparable to simple forwarding and a latency profile that is two to three times faster than NFV. In addition, we present several use-cases that illustrate the capabilities of DPX's security actions in detecting and responding to network attacks and 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 by over 60%.

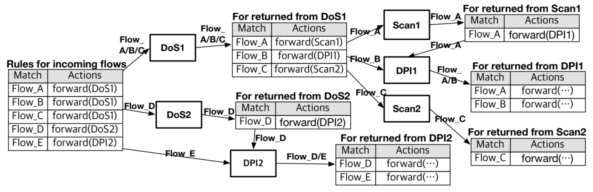
2 Motivating Challenges

With the increased adoption of SDN, SDN- and NFV-based security solutions are also gaining in popularity. However, these are associated with the following challenges.

Performance and Management Challenge: Figure 1 illustrates two possible strategies for deploying NFV-based security services. First, security services can be deployed as an SDN application on a control plane and several applications are already designed to provide security functionalities. This approach has the advantage of being easily adaptable and manageable, but it is difficult to use practically due to many constraints. Because of the architectural limitations of SDN (e.g., difficulty in inspecting packet payloads), sophisticated security services cannot be implemented without third party applications or devices. Moreover, a centralized controller must handle all underlying network packets for security services, resulting in a significant performance overhead on the controller. This issue has been also discussed by *Yoon et al.* [36]; this work demonstrates that using SDN applications for security services causes serious overhead in many cases.



(a) Service chains example



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 to handle the service chains

Fig. 2. An illustration of the flow-steering challenge in service chaining.

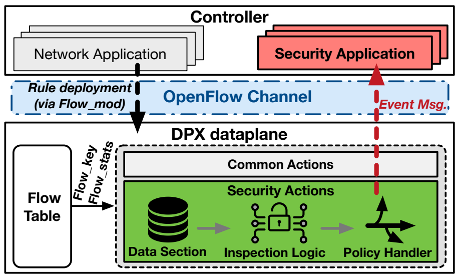
Alternatively, security services can be deployed as virtualized network functions (VNFs) on middleboxes (i.e., NFV). It allows deploying complicated security services without functional limitations. However, since a middlebox is located remotely, this approach would imply that all network traffic should take extra hops that deviate from the shortest path to the destination, resulting in performance degradation for both latency and bandwidth. For example, when a packet is to be transmitted from the switch A to B in Figure 1, the shortest distance is one hop, but it is stretched to three hops if it goes through the middlebox. Therefore, the latency is increased and the network bandwidth is wasted due to the extended path. In addition, since the VNFs operating in the middlebox are independent instances, an administrator should arrange extra control channels for each VNF.

Flow-Steering Complexity Challenge: As network threats become more sophisticated, a single security device alone is not enough to secure a network. Hence a multi-prong approach to security is adopted using SDN to compose service chains that integrate multiple security features in a series, allowing each packet to be investigated by multiple services. However, there is an operational challenge to configure such service chains in the data plane. For instance, we assume that we configure service chains for five flows with different service chains according to security policy as shown in Figure 2a. To operate these service chains, complicated flow rules, as shown in Figure 2b, are required in the SDN data plane. Each flow is forwarded to the designated security instance, and the security instance returns the flow to the data plane after inspection. Then, the data plane forwards the flow to the next security instance.

The flow-steering challenge is closely related to challenges associated with management and troubleshooting of a network. As detecting network faults is tedious [27, 35], simplified network topologies are preferred to mitigate misconfigurations. While efficient service chain design has been extensively studied [6, 26], little attention has been paid about the accompanying issues that arise while constructing service chains. In particular, several research efforts [5,37] have studied the prevalence of network misconfigurations whose likelihood will be further exacerbated by the need for complicated traffic steering rules.

3 System Design

To address the challenges discussed, we devise a novel data plane extension for SDNs, called DPX, that provides security functions as part of the packet processing logic.



(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. 3. Illustration of DPX Design and Processing Workflow

3.1 DPX **Overview**

Figure 3 illustrates the overall design of DPX and its workflow. DPX defines security functions in terms of one or more actions following the OpenFlow protocol paradigm [21], which handles network packets with pre-defined actions using the match-action interface. These DPX *security actions* perform security services on incoming packets as part of the packet processing in a dataplane. For instance, in the flow table of Figure 3b, the actions for `Flow_A` will monitor network flows to detect whether flows send/receive more than 1000 Mbps, and the actions for `Flow_B` will perform multiple network security functions (i.e., vertical scan detector, session monitor, and deep packet inspector) on the corresponding traffic. The security actions can be enforced with the OpenFlow protocol from DPX applications running on an SDN controller.

3.2 DPX **Actions**

Each security action is an individual packet processing block in the DPX dataplane. After looking up a matched flow rule in the flow tables for an incoming packet (i.e., parsing packets, looking up flow entries corresponding to the packets, and updating flow statistics), DPX runs actions in the matched rule. In this case, if the list of actions includes a security action(s) during execution, DPX will trigger a corresponding security action block for the matched flow, and the security action performs a designated security check, through the following three steps, as depicted in Figure 3a.

First, DPX updates metadata or statistics of packets used for inspection into the *data section* using (1) the *flow_key* which stores the packet-level metadata used for indexing flow tables, and (2) the *flow_stats* which contains the statistical data (e.g., the count and bytes of packets) of each flow entry. 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.

Second, the *inspection logic* performs an actual security operation to a packet within the data section. For instance, the inspection logic for 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), and the inspection logic for 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. The result of the operation is compared with a threshold set by a user to decide whether or not to violate a security specification.

Third, if there is a security violation, DPX handles packets according to one of the three policies: (1) `alert` which sends an alert message to a controller with a datapath ID, physical port number associated, the reason for event occurrence, reference features (e.g., the current bps), raw packet data, and a cluster ID (we will describe this in Section 3.3); (2) `discard` which terminates the packet processing sequence and drops a detected packet; and (3) `redirect` which forwards packets to alternative destinations (e.g., honeypot) instead of the original destination.

A DPX action is configured by its parameters like common OpenFlow actions (e.g., `set_nw_src(10.0.0.1)`), and the parameters must be set when a security action is installed. For example, a bps threshold and a pattern list are the required parameters for the DoS detector and the deep-packet inspector respectively. Depending on the type of security actions, there may be one or more parameters, including the policy parameter describing how to handle a detected packet. Specifically, we can represent the security policy "if a 1000 Mbps DoS attack is detected, redirect the following traffic to port 2" as "`sec_dos(mbps=1000,policy=redirect:2)`".

Benefits of DPX **actions.** We describe some noteworthy benefits of DPX actions.

1) Fine-grained security deployment: First, DPX actions integrate security operations to a flow steering, so that a network administrator can only focus on a flow direction (e.g., whether normal traffic reaches its destination) without considering security configurations. This *fine-grained security deployment* simplifies the management issue by reducing the number of flow rules to detouring middleboxes.

2) Simplified service chains: Second, DPX can compose a service chain in the simplified method. Regardless of the complexity of the service chain and the number of services, a single line of a flow rule can represent the service chain by enumerating the actions. For example, if we configure a service chain for a *DoS detector, an anomaly detector, a vertical-scanning detectors, a session monitor, and a payload inspector* of a flow destined to a 10.0.0.1 host, it can be described through a single rule as follows:

```
Flow: ...,nw_dst=10.0.0.1,...,actions=sec_dos(...),
sec_anomaly(...),sec_vscan(...), sec_session(...), sec_dpi(...),...
```

3) Optimized processing sequence: Another benefit is that DPX optimizes packet processing by eliminating unnecessary traffic steering to NFVs or middleboxes. In the case of a conventional SDN/NFV environment, traffic has to be detoured to an NFV host before reaching its destination. Therefore, the total traversal distance of the traffic is stretched, and the propagation delay and the bandwidth usage are increased as much as the detoured path.

In addition, this detouring involves the latent wasting of network resources by redundant packet processing. Generally, a switch forwards an incoming packet in four steps (i.e., parse packets, look up flow tables, update flow stats, and execute actions), and if security inspection is required for the packet, it would be forwarded to an NFV host. After receiving the packet, the NFV host examines the packet through parsing, classifying, updating, and inspecting, similar to a switch. Then, the NFV host returns the packet to the switch, and the switch takes the packet processing steps *one more time* to forward the packet to its original destination. Namely, the packet detouring naturally connotes a redundant packet process, and this leads to a waste of network

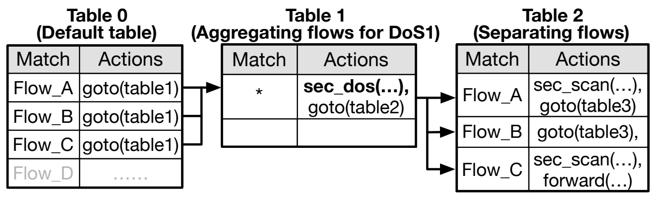


Table 0 (Default table)
Match
Actions
Flow_A goto(table1)
Flow_B goto(table1)
Flow_C goto(table1)
Flow_D ......
Table 1 (Aggregating flows for DoS1)
Match
Actions
* sec_dos(...), goto(table2)
Table 2 (Separating flows)
Match
Actions
Flow_A sec_scan(...), goto(table3)
Flow_B goto(table3),
Flow_C sec_scan(...), forward(...)

Fig. 4. Traffic steering with security actions

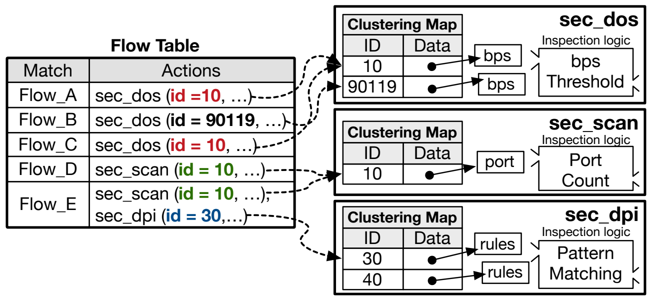


Fig. 5. Design of Action Clustering

resources to degrade network throughput. We have checked that this performance degradation indeed exists (See Section 6). For a service chain, such inefficient operations would be repeated multiple times and worsen the performance.

In the case of DPX, security services are supported on a switch directly, so traffic does not need to be detoured, and the latency and bandwidth loss would be avoided. Also, security actions process traffic in a manner that minimizes redundancy: each security action utilizes the *flow_key* and the *flow_stats* generated in the flow lookup step, rather than incurring a separate packet analysis process. Packets already contain appropriate information such as addresses, protocol, and packet count and size, so the overhead incurred by the security actions will be practically bounded to performing the inspection logic – even in a service chain.

3.3 Action Clustering

Although DPX actions address the challenges raised in Section 2, there remain additional complexities and opportunities for optimization; particularly when internal traffic-steering policies dictate that service chains share the same monitoring instance between multiple flows. For example, when we consider Figure 2a with DPX security actions, Flows A, B, and C for DoS1 would be expressed as Figure 4; because each individual flow shares the same DoS monitor, Flows A,B, and C are redirected to the next table for aggregated DoS tracking. After executing the DoS action, those flows are forwarded to the next table for additional actions.

To address this redundant traffic steering challenge, we propose a novel technique called the *action clustering* which merges DPX actions of multiple flow rules into a few synthetic rules. Figure 5 illustrates the workflow of action clustering. DPX builds the clustering map, per DPX action, which consolidates and manages the data section based on the cluster ID. The cluster ID is a DPX action parameter used as the hash key to lookup the clustering map before the data segment is updated and delivered to the

Match	Actions
Flow_A	sec_dos(**id=10**, ...), sec_scan(**id=10**, ...),sec_dpi(**id=10**, ...), ...
Flow_B	sec_dos(**id=10**, ...), sec_dpi(**id=10**, ...), ...
Flow_C	sec_dos(**id=10**, ...), sec_scan(**id=20**, ...), ...
Flow_D	sec_dos(**id=20**, ...), sec_dpi(**id=20**, ...), ...
Flow_E	sec_dpi(**id=20**, ...), ...

Fig. 6. Simplified flow rules with action clustering

inspection logic. The shared data segment facilitates detection of abnormal behavior not just within a given flow but also abnormal behavior across aggregated flows. For example, in Figure 5, `Flow_A` and `Flow_C` have the same action ("sec_dos") and the same cluster ID (10). Thus, the aggregated data for both flow rules are maintained in the clustering map for DoS detector. When the packets of `Flow_A` and `Flow_C` arrive at the switch respectively, the statistics (in this case, the packet length) of the flows are accumulated and updated for each flow. If a DPX action runs standalone, its cluster ID would be set with a unique random ID (`Flow_B`). Since DPX considers the type of actions and the cluster ID together, the same cluster ID between different actions does not correlate (e.g., the DoS detector action for `Flow_A/C` and the scanning detector action for `Flow_D` which use the same cluster ID 10). The action clustering works regardless of service chaining. Even when a service chain is configured, the clusters in the chain drive independently for each action (`Flow_E`).

Here, we address the motivational challenge as shown in Figure 2a. With action clustering, an administrator could configure security service chains as shown in Figure 6. In this service chaining, the flows share the same cluster ID between the same instances (i.e., DoS1 for flow A/B/C, DPI1 for flow A/B, and DPI2 for flow D/E).

3.4 Advanced Action Clustering

In this section, we introduce two additional features, *inconsistent clustering* and *multi-clustering* which make the action clustering more flexible in practice.

1) Inconsistent-Parameter Clustering. In some circumstances, different detection policies may need to be applied to a traffic stream, e.g., stringent security control on some flows and loose security on others. *Inconsistent-parameter clustering*, which takes different parameters to allow each clustered action to react differently under the shared data section, can be used in this scenario. The following flow rules serve as examples.

```
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 are in the same cluster, they share the same data that contains the count of port hits and the last arrival time of each port, but each `sec_vscan` has different parameters for detecting different scanning attacks. In this case, those parameters imply that multiple security policies are applied to the same data. For instance, when the current aggregated count of port hits is 700 in the last three seconds and 900 in the last five seconds, DPX only triggers an alert against `Flow_B` and not `Flow_A`.

2) Multi-Clustering. The key idea behind multi-clustering is that 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 incoming flow state (i.e., flow_key and stats) with all related clusters. Thus, multi-clustering allows a flow to be enforced by a sub security policy alongside a parent security policy. For example, it is possible to set an additional bandwidth limitation to a flow of interest while preserving the original bandwidth limitation.

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

The DoS action for `Flow_B` involves two cluster IDs, 10 and 20, but `Flow_A` is only associated with cluster 10. This means that the DoS action monitors the bandwidth for both flows using ID 10, while separately monitoring the bandwidth for Flow_B using ID 20. It implies that the two flows (Flow_A and B) should not exceed the total of 1000 Mbps, and also Flow_B, by itself, should not exceed 500 Mbps (but Flow_A can reach up to 1000 Mbps).

3.5 Action Enforcement

Since security functions are designed as parts of actions, DPX applications that engage security actions can be implemented on top of an SDN controller. By appending DPX actions into `Flow_MOD` messages, an administrator can deploy security functions to a specific network flow. Then, a DPX switch will execute the security functions defined in the action fields by the Flow_Mod messages. When security actions on the data plane detect abnormal behaviors (e.g., any violation of DoS thresholds) with the alert policy, it sends an alert, which is an OpenFlow message with the detection information, to the SDN controller. Then, the DPX application in a controller receives the alert message and responds with appropriate reactions to abnormal behaviors by registering its event handlers.

In addition, DPX preserves the same workflow as the original OpenFlow interface. Therefore, not only is it highly compatible with the existing OpenFlow API and features, administrators with experience in OpenFlow can also integrate DPX on their network with minimal effort and overhead.

4 System Implementation

To validate our design principles, we developed a prototype implementation of DPX in both software and hardware. We provide details on both implementations below.

4.1 **Software-based** DPX

The software-based DPX switch is implemented based on Open vSwitch (OVS) [20,24] version 2.4.9, as shown in Figure 7. In order to perform actions on a matched packet from the flow table, OVS will invoke the `execute_actions` sequence with packet data (i.e., socket buffer (SKB)) and an action key which enumerates a list of actions to be executed for the matched flow. We modified the execute_actions module of OVS to call

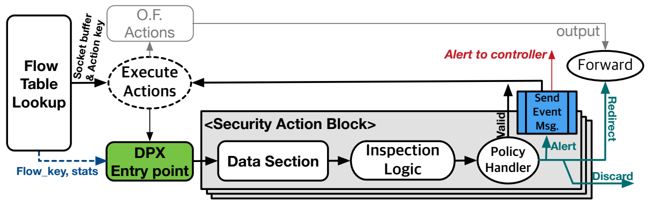


Fig. 7. An illustration of the software-based DPX datapath

the `DPX entry point` that is the starting point for DPX security actions when the flow key includes security actions. A security action is composed as a modular function block which is a function interface block performing three stages described in the previous section (i.e., update the data section, perform the inspection logic, and impose the policy on detected traffic), and a new security action can be added by registering a new block to the DPX entry point.

To demonstrate this implementation, we present six DPX security action blocks in the software-based DPX: (1) DoS detector which detects a bandwidth exceed by Mbps threshold, (2) Deep Packet Inspector (DPI) which finds a matched pattern in a packet payload, (3) Anomaly detector which detects a change rate of bandwidth, (4) Vertical-Scanning (vScan) detector which counts how many TCP/UDP ports are hit within a time window, (5) Horizontal-Scanning (hScan) detector which counts how many hosts for a specific TCP/UDP port are hit within a time window, and (6) Session monitor which traces TCP sequences and counts invalid connections. The total number of supported flows with an action cluster is only limited by the memory capacity of a host device.

4.2 **Hardware-based** DPX

The hardware-based DPX switch is implemented using the NetFPGA-SUME board which is an FPGA-based PCI Express board with four SFP+ 10 Gbps interfaces [38]. We migrated the OpenFlow IP package of NetFPGA-10G board [22] to our NetFPGA-SUME board and extended it to support DPX actions. To enable DPX, the hardware-based DPX includes the security processing sequence that consists of a security action selector, security action modules, and a policy handler as depicted in Figure 8.

After looking up the flow table, a flow_key, flow_stats, and an action key are delivered to the security action selector, which looks up security actions to be executed by the action key. A security action in the hardware-based DPX is designed as a security action module which is an independent entity that contains a data section with its own memory space. Thus we can easily add new features by registering a new security module to the security action input selector. When security actions are executed, all security action modules will be executed in *parallel*. Therefore, the security action input selector transfers all parameters related to security actions through the wide data bus, and packet data are carried to each security module from the packet buffer, as shown in Figure 8. Then, when a security violation is detected among the performed security actions, the policy handler applies a designated policy to the detected packet. If a security violation is detected by multiple security modules and policies are con-

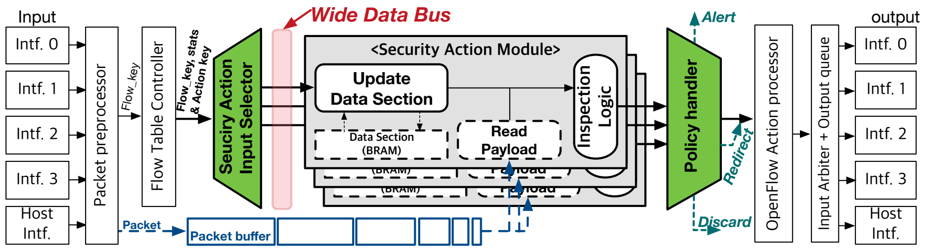


Fig. 8. An illustration of components and dataflow in the hardware-based DPX

flicted, a higher priority policy is applied; the priority of policies is *redirect* → *discard* → *alert* (high to low).

We verify the hardware-based DPX by designing two security action modules: (1) DoS detector with 1024 action clustering blocks, and (2) DPI action with four action clustering blocks that can each store 1024 patterns of 256 bytes. The number and length of rules can increase depending on memory configuration in NetFPGA-SUME.

4.3 Controller and OpenFlow protocol

To enable enforcement of security actions, we have designed an application programming interface for the POX controller [25] which is a Python-based SDN controller. We have supplemented the DPX event handler class to receive DPX messages and implemented a new Python module supporting DPX applications. We have added around 500 lines of Python code to POX to enable all DPX related functions. Finally, we have used the OpenFlow 1.0 vendor extension for communication between the control plane of DPX (i.e., POX implementation) and the DPX switch. It is worth noting that our design principles will also apply to more recent versions of OpenFlow and modern controllers such as ONOS [2] or OpenDaylight [14]. We chose POX and OpenFlow 1.0 due to their simplicity for rapid prototype development.

5 Security Use Cases

To highlight operational use-cases of DPX, we first set up a testbed (shown in Figure 9), where a DPX switch is used to connect a malicious host and a server (hosting FTP service, which has a buffer-overflow vulnerability). The switch is controlled by a controller running DPX security applications. A malicious host is used to perform three different attacks: (i) DoS, (ii) port scanning, and (iii) buffer overflow 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 assumed that a network administrator has pre-configured the security applications for reacting to network attacks.

1) Denial of Service. DPX has two mechanisms to detect DoS attacks: DoS detector actions and anomaly detector actions. In this example, we employ a DoS detector action to alert when the traffic surpasses 500 Mbps, and the malicious host sends over 1 Gbps traffic to the FTP server using hping3 [8]. When the DoS detector action detects the attack (i.e., high-volume traffic), it sends an alert message including current packet-rate information to the DPX controller (Figure 10a). Then, the DPX application

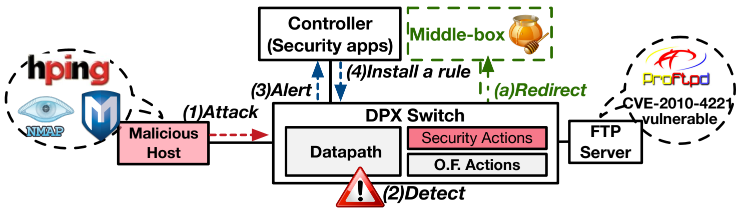
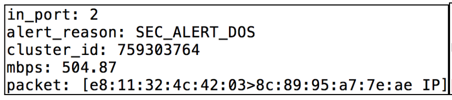
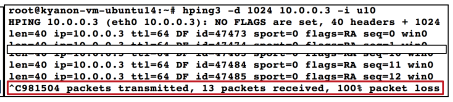


Fig. 9. Testbed and Operational Scenario of DPX Use Cases



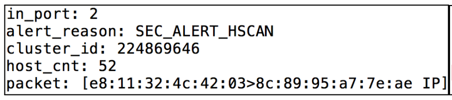
```
in_port: 2
alert_reason: SEC_ALERT_DOS
cluster_id: 759303764
mbps: 504.87
packet: [e8:11:32:4c:42:03>8c:89:95:a7:7e:ae IP]
```

(a) DoS attack alert message



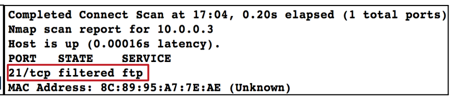
```
root@kyanon-vm-ubuntu14:~# hping3 -d 1024 10.0.0.3 -i u10
HPING 10.0.0.3 (eth0 10.0.0.3): NO FLAGS are set, 40 headers + 1024
len=40 ip=10.0.0.3 ttl=64 DF id=47473 sport=0 flags=RA seq=0 win0
len=40 ip=10.0.0.3 ttl=64 DF id=47474 sport=0 flags=RA seq=1 win0
len=40 ip=10.0.0.3 ttl=64 DF id=47483 sport=0 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 the malicious host



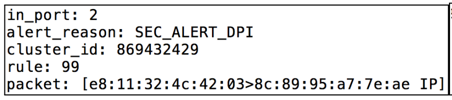
```
in_port: 2
alert_reason: SEC_ALERT_HSCAN
cluster_id: 224869646
host_cnt: 52
packet: [e8:11:32:4c:42:03>8c:89:95:a7:7e:ae IP]
```

(c) Horizontal scanning alert message



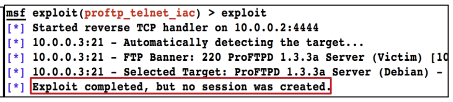
```
Completed Connect Scan at 17:04, 0.20s elapsed (1 total ports)
Nmap scan report for 10.0.0.3
Host is up (0.00016s latency).
PORT   STATE    SERVICE
21/tcp filtered ftp
MAC Address: 8C:89:95:A7:7E:AE (Unknown)
```

(d) Nmap result in the 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 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 the malicious host

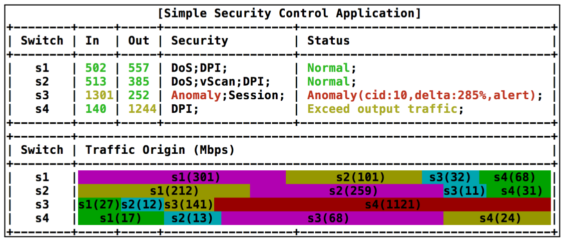
Fig. 10. Alert messages & block results for various use-case attacks

installs a new flow rule to block the attack traffic in DPX. Hence, most of the traffic is dropped, and the DoS attack is suppressed as shown in Figure 10b.

2) Port Scanning. DPX can detect horizontal and vertical scanning via its scan detector actions. In addition, the session monitor action can help in detecting stealth scanning attacks. In this example, we configure the horizontal scan detector action with 50 hosts and a 10 second time window. The malicious host generates horizontal scans directed at port TCP/21 using nmap [18]. When DPX successfully detects the horizontal scanning, DPX sends an alert message including current host count to the controller (Figure 10c). Then, the DPX application installs a new flow rule to block the attack traffic, as shown in Figure 10d.

3) Remote Exploit. Next, we consider the case where a malicious host tries to exploit the vulnerability of ProFTPD to get a remote shell. In this use-case, the malicious host uses Metasploit [16] which is a very popular penetration testing tool. This class of attacks can be detected by the DPI action, so we set the DPI action with 100 rules including the attack pattern at 99th. After performing the attack, the 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 (Figure 10e). 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 Figure 10f.

4) Middlebox Cooperation. DPX can cooperate with other middleboxes using the `redirect` policy to redirect packets, such that middleboxes or NFV services only pro-



```
                    [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. 11. Simple security control with DPX

cess packets filtered by DPX, instead of all packets. For example, the network in Figure 9 runs a honeypot with security actions on the DPX switch. When a security action has the *redirect* policy, benign connections are forwarded to the original destination, but only suspicious connections are classified and transmitted to the honeypot. In addition to this approach, this *conditional packet handling* can be applied to implement other network security solutions such as honeynets or Moving Target Defense.

5) Security Control. Since DPX is designed to be compatibile with OpenFlow, a security control solution can be implemented on a controller by combining the network management ability of OpenFlow and DPX security features. Figure 11 is the example application that is built on the POX controller; It collects information about the switches by requesting 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. Using those collected information, the application displays the direction and amount of traffic between each switch and the current security status. If a security violation occurs, the administrator can establish a future security policy based on observed network conditions. For example, in the case of Figure 11, the anomaly detector action of the switch `s3` alerts that current traffic-level is 285% higher than usual. A 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`. Then, the administrator can block traffic for `s4` to `s3`, or deploy stricter security actions to defend against future attacks.

6 System Evaluation

The test environment 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, as illustrated in Figure 12. All machines run Ubuntu 14.04 and have an Intel Xeon E5-2630@2.9GHz processor, 64 GB of RAM, and Intel X520-DA2 10GbE NICs. The datapath device uses the NetFPGA-SUME board for running the hardware-based DPX switch, or runs the software-based DPX switch on the host OS. Although DPX may be deployed in multi-switch environments, we focused on single switch evaluations because the overall throughput is determined by the bottleneck switch.

We measured the throughput and latency of DPX actions in three different ways to verify the impact of DPX actions on performance. First, we configured the DPX switch to forward all incoming packets from `h1` to `h2` after passing DPX actions (e.g., `in_port=1,actions=sec_dos(...),output:2`). Second, to compare the performance

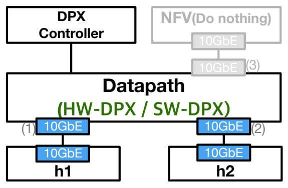
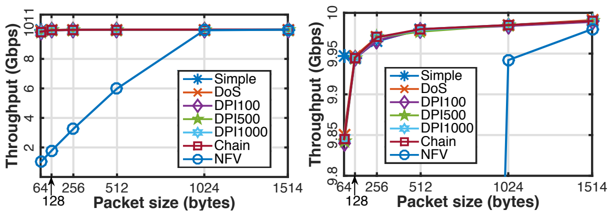


Fig. 12. Evaluation Topology



(a) Full-size graph (b) Magnified graph

Fig. 13. Throughput of Hardware-based DPX

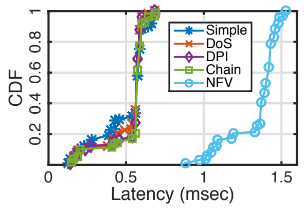
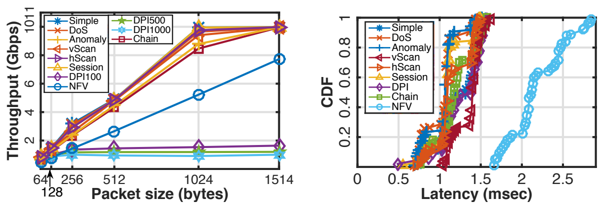


Fig. 14. Latency of HW-DPX



(a) Throughput (b) Latency

Fig. 15. Performance of Software-based DPX

overhead of DPX security services, we measured the simple forwarding delay on the native NetFPGA-SUME and Open vSwitch 2.4.90 from `h1` to `h2` without any special handling (i.e., `in_port=1,actions=output:2`). Finally, to evaluate how much DPX improves the performance compared with a conventional NFV environment, we measured the performance when packets traverse an NFV host before arriving at `h2` (i.e., `in_port=1,actions=output:3` and `in_port=3,actions=output:2`). The NFV host does nothing and immediately returns packets to the DPX switch. Here, please note that we do NOT mean to measure the performance of each security action or verify their functionality. Our evaluation aims to measure the overhead of processing security functions as security actions in the datapath and how much performance improvement DPX provides over NFVs. The throughput was measured with various size of packet bursts generated by Intel DPDK-Pktgen [10], and the latency was measured through the RTT of TCP packets that contain random 256-byte payloads generated by nping [19].

6.1 Evaluating Hardware-based DPX Performance

Throughput. Figure 13a illustrates the throughput of the hardware-based DPX. All security actions (i.e., DoS detector and DPI with 100, 500 and 1000 rules) achieved throughput close to 10 Gbps for the simple forwarding case. On closer inspection, (Figure 13b), while we incur minimal overheads ($< 1\%$) for the worst case of 64-byte packet burst, it is negligible and the line-rate performance is realized for bursts of higher-packet sizes. When configuring the security service chain with DoS and DPI, we find that there is no observable overhead because of the parallel processing provided by hardware-based DPX.

We also note that the NFV-based approach incurs significant overheads before the 1024-byte point. In particular, it delivers only 1 Gbps of throughput at the 64-byte point. This degradation is mainly caused by the bottleneck on the NFV host and processing overhead of the incoming and outgoing packet stream. While this throughput degradation can be moderated through improvements of the NFV host itself, it is difficult to alleviate the bottleneck completely considering that multiple virtual machines could potentially be service-chained in actual NFV deployments.

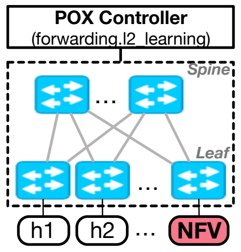
Latency. The CDF in Figure 14 illustrates the latency induced by hardware-based DPX switch processing. In the most cases, the latency of DPX actions converges to the latency of simple OpenFlow forwarding; 99% of packets are processed in less than 0.65 ms. Even in the case of the service chains, there is no meaningful overhead in latency. This result is remarkable when we compare DPX actions with the NFV host. Even if the NFV host directly returns traffic without any additional processing, the latency is a factor of two or more times higher than DPX actions.

6.2 Evaluating Software-based DPX Performance

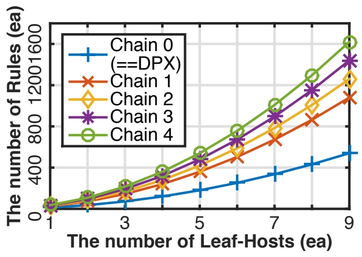
Throughput. Figure 15a presents the throughput results of software-based DPX. Most of the DPX actions, except the DPI action, incur small overheads compared to the simple forwarding of the native OVS; In all byte ranges, DPX services achieve at least 90% of the throughput of simple forwarding. In the case of composite service chaining that comprises of all security features except DPI (i.e., DoS detector, Anomaly detector, Vertical/Horizontal Scanning Detector, and Session Monitor), there is only a minor performance degradation, that is comparable to the overhead of just using the anomaly detector. This performance degradation is not the overhead by the chaining itself, but mainly the bottleneck by the worst performing security action in the chain.

These results indicate that DPX offers a compelling performance improvement over the NFV-based solutions. The NFV-based approach only achieves a throughput of 7.7 Gbps in the best case (i.e., 1514-byte point). Here, considering that the simple forwarding of both software and hardware gets nearly 10 Gbps at the 1514-byte point, the throughput of the NFV detouring should also be similar in both the software and the hardware cases at the 1514-byte point because detouring to NFV is based on simple forwarding. However, the software-based NFV detouring has lower throughput. This means that there are other performance degradation factors besides the bottleneck on the NFV host; this extra overhead is due to OVS having to concurrently process two packet streams since the NFV host returns a packet stream to OVS and `h1` continues to send the packet stream to OVS for a certain period. Thus the bandwidth capacity of OVS is exceeded causing it to underutilize the bandwidth of the network. This overhead was less obvious on the hardware switch, but the throughput limitations of the software environment clearly exposes the bandwidth wastage from NFV detouring.

An exception to this is the DPI action, which barely achieves 1 Gbps, with throughput decline that follows the number of rules (i.e., 100, 500 and 1000 rules). This limitation is due to the overhead of pattern matching in software; both Snort [32] and Suricata [34] in the IPS mode provide similar throughputs of 0.8-1.2 Gbps in our tests. Although it is difficult to objectively compare the DPI action with Snort and Suricata because of functional differences, this result suggests that the DPI action could be utilized for the initial inspection at edge nodes before forwarding to native DPI instances for improved performance.



(a) Leaf-Spine Topology



(b) Count of required rules

Fig. 16. Evaluating DPX Flow Table Simplification

Latency. Figure 15b illustrates the latency of the software-based DPX as a CDF graph. In most cases, the latency of DPX actions approaches that of the latency of simple forwarding in native OVS including the DPI action; 99% of packets, are processed in less than 1.5 ms. We cannot also find any meaningful overhead while constructing service chaining. Therefore, we can see that there is very little processing time delay while configuring service chaining with DPX. Here, NFV detouring incurs about twice the latency of the hardware case. We also verified that the extra overhead was caused by the propagation delay to the NFV host on the switch; In the case of simple forwarding, the average latency of software-based switching is 1.21 ms and hardware-based switching is 0.50 ms. However, the average of latency for the NFV detouring case with software-based switching is 2.18 ms and for the hardware-based switching is 1.34 ms. In summary, while hardware-based switching increased latency by 37%, the overhead of software-based switching is 55%. This difference could be attributed to 20% extra overhead in the path expansion latency and redundant packet processing at the switch. Finally, we also measured the computational overhead of DPX actions, and it was negligible (1-2% of the packet-switching overhead).

6.3 Flow Table Simplification

We conducted an evaluation of the effectiveness of DPX in simplifying flow rules. However, since flow tables vary depending on the controller (application) configuration, network policy and background traffic, it is difficult to make universal claims. Hence, we have assumed a specific use case and emulated it using Mininet [12] (a popular virtual network emulation tool) and the POX controller. While we make no claim as to the representativeness of the use case or topology, we believe that qualitatively similar benefits could be extended to real NFV networks.

We emulated a leaf-spine topology that is a two-layer datacenter network architecture, and connected end hosts to each leaf switch as depicted in Figure 16a. 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.

As shown in Figure 16b, 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.

7 Discussion

Our evaluation demonstrates that DPX imposes minimal overhead that approaches the line-rate performance of simple forwarding with respect to both throughput and latency. We further find that the performance degradation of NFV actually stems from a variety of auxiliary overheads such as packet re-processing on the NFV and steering overhead on the switch that contribute to the overall reduction in throughput, beyond added path latency. In a real network, a service chain may be lengthened thereby requiring additional hops to the NFV device. In such cases, the resulting degradation will be greater than in our experimental environment.

DPX can also reduce the number of flow rules including data-plane security functions, and the action clustering allows complex security policies to be expressed in a simplified flow table as described in Sections 2 and 3. Today, networks with thousands of hosts are commonplace, thanks to virtualization technology. Hence, optimizing the network traffic engineering in terms of both performance and security while considering the impact of NFV placement places an onerous challenge on network administrators. We believe that describing complex security policy in terms of aggregate action clusters will help to relieve configuration, management and debugging workloads. In practice, DPX enables the network administrator to prioritize traffic engineering over security-device routing and service-chain management.

Limitations. Since DPX operates security services at each switch, DPX cannot directly enable distributed solutions that span the whole network, such as network-wide DoS detection. This can be addressed by adding support for controller-based applications, as discussed in the use-cases section. Another potential solution involves designing a DPX data-exchange protocol between security actions on a network. In future work, we plan to investigate new message protocols that exchange network statistics (e.g., current bandwidth, number of active flows) across DPX switches to synchronize security information and enable fully distributed security-policy enforcement.

8 Related Work

Our paper is informed by prior work on network-function control and data-plane-based network functions. The CloudWatcher [30] service uses OpenFlow to detour network flows to physical network devices in dynamic cloud networks, where the security functions are pre-installed. SIMPLE [26] and Gupta *et al.* [6] propose efficient traffic steering for composing service chaining with middleboxes. CoMb [28] consolidates network middleboxes into a single physical machine to reduce capital expenses and device sprawl. While these systems streamline NFV deployment, they do not address the associated degradation challenge due to NFV detouring.

Popular software switch implementations [7, 20, 24] enhance scalability by supporting a large number of virtual ports to guarantee high performance. Pisces implements a custom software switch using the P4 processing language [29] and SwitchBlade proposes the use of FPGAs to build custom network protocols on hardware [1]. AccelNet [4] uses FPGA-based smart NICs to support the bandwidth needs of network datacenters. OFX [33] is an OpenFlow extension framework which enables an OFX application to be loaded onto a switch at a runtime. NEWS [15] is an extended SDN architecture that handles packets through modified switch flow tables called app tables. Their approach results in redundant processing sequences for network services as these are implemented as external modules. In contrast DPX integrates all security functionaliity inside the SDN data plane and thus its performance overhead is minimal. QoSE [23] is another data plane module that provides security functions using a distributed NFV, but it suffers from performance degradation due to frequent traffic detouring. Avant-guard [31] uses data-plane connection migration and actuating triggers to defend against control-flow saturation attacks. However, its objectives are narrower and it does not afford the flexibility to run arbitrary security services.

9 Conclusion

This paper presented the design and (hardware/software) prototype implementations of a new data plane architecture called DPX, which embeds security functions directly into a switch as a set of actions. DPX simplifies composition of service chains and enables graceful integration of security services. DPX is carefully engineered to mitigate associated detouring overheads, such that it can provide security services with maximal performance, producing a performance profile that is comparable to simple forwarding and a latency profile that is two to three times faster than traditional NFV. Furthermore, action clustering reduces the number of flow rules and eliminates unnecessary service-chaining actions by aggregating them into a single DPX action. As the network becomes more complex, we expect that the approach of DPX has high-potential that could be utilized not just to support efficient academic security projects, but also industrial operations.

Acknowledgement. KAIST was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2018-0-00254, SDN security technology development). SRI International was supported by the National Science Foundation(NSF) award no. 1642150.

References

1. Anwer, M.B., Motiwala, M., Tariq, M.b., Feamster, N.: Switchblade: a platform for rapid deployment of network protocols on programmable hardware. ACM SIGCOMM Computer Communication Review **40**(4) (2010)

2. Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., Koide, T., Lantz, B., O'Connor, B., Radoslavov, P., Snow, W., et al.: Onos: towards an open, distributed sdn os. In: Proceedings of the third workshop on Hot topics in software defined networking. pp. 1–6. ACM (2014)
3. Fayaz, S.K., Tobioka, Y., Sekar, V., Bailey, M.: Bohatei: Flexible and elastic ddos defense. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 817–832. USENIX Association, Washington, D.C. (Aug 2015), `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/fayaz`
4. Firestone, D., Putnam, A., Mundkur, S., Chiou, D., Dabagh, A., Andrewartha, M., Angepat, H., Bhanu, V., Caulfield, A., Chung, E., et al.: Azure accelerated networking: Smartnics in the public cloud. In: 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), Renton, WA (2018)
5. Gill, P., Jain, N., Nagappan, N.: Understanding network failures in data centers: measurement, analysis, and implications. In: ACM SIGCOMM Computer Communication Review. vol. 41, pp. 350–361. ACM (2011)
6. Gupta, A., Habib, M.F., Mandal, U., Chowdhury, P., Tornatore, M., Mukherjee, B.: On service-chaining strategies using virtual network functions in operator networks. Computer Networks **133**, 1–16 (2018)
7. Honda, M., Huici, F., Lettieri, G., Rizzo, L.: mswitch: A highly-scalable, modular software switch. In: Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research. pp. 1:1–1:13. SOSR '15, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2774993.2775065, `http://doi.acm.org/10.1145/2774993.2775065`
8. hping3: A network tool able to send custom TCP/IP packets and to display target replies, `http://www.hping.org/hping3.html`
9. Hwang, J., Ramakrishnan, K.K., Wood, T.: Netvm: High performance and flexible networking using virtualization on commodity platforms. In: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). pp. 445–458. USENIX Association, Seattle, WA (Apr 2014), `https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/hwang`
10. Intel: Intel DPDK: Data Plane Development Kit, `http://dpdk.org`
11. Kim, H., Feamster, N.: Improving network management with software defined networking. IEEE Communications Magazine **51**(2), 114–119 (2013)
12. Lantz, B., Heller, B., McKeown, N.: A network in a laptop: rapid prototyping for software-defined networks. In: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks. p. 19. ACM (2010)
13. Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., Huici, F.: Clickos and the art of network function virtualization. In: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). pp. 459–473. USENIX Association, Seattle, WA (Apr 2014), `https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins`
14. Medved, J., Varga, R., Tkacik, A., Gray, K.: Opendaylight: Towards a model-driven sdn controller architecture. In: World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on a. pp. 1–6. IEEE (2014)
15. Mekky, H., Hao, F., Mukherjee, S., Lakshman, T., Zhang, Z.L.: Network function virtualization enablement within sdn data plane. In: IEEE INFOCOM. pp. 1–9 (2017)
16. Metasploit: Penetration Testing Software, `https://www.metasploit.com/`
17. NetFPGA: NetFPGA-SUME board, `https://netfpga.org/site/#/systems/1netfpga-sume/details/`
18. nmap: Network Mapper - Security Scanner, `https://nmap.org/`
19. Nping: An Open source network packet generation,, `https://nmap.org/nping/`

20. Open vSwitch: An Open Virtual Switch, http://openvswitch.org/
21. OpenFlow: Open network foundation, https://www.opennetworking.org/sdn-resources/openflow
22. Organization, N.G.: Netfpga 10g openflow switch (2012), https://github.com/NetFPGA/NetFPGA-public/wiki/NetFPGA-10G-OpenFlow-Switch
23. Park, T., Kim, Y., Park, J., Suh, H., Hong, B., Shin, S.: Qose: Quality of security a network security framework with distributed nfv. In: Communications (ICC), 2016 IEEE International Conference on. pp. 1–6. IEEE (2016)
24. Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., Amidon, K., Casado, M.: The design and implementation of open vswitch. In: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15). pp. 117–130. USENIX Association, Oakland, CA (May 2015), https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff
25. POX: Python Network Controller, http://www.noxrepo.org/pox/about-pox/
26. Qazi, Z.A., Tu, C.C., Chiang, L., Miao, R., Sekar, V., Yu, M.: Simple-fying middlebox policy enforcement using sdn. In: Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM. pp. 27–38. SIGCOMM '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2486001.2486022, http://doi.acm.org/10.1145/2486001.2486022
27. Roy, A., Zeng, H., Bagga, J., Snoeren, A.C.: Passive realtime datacenter fault detection and localization. In: NSDI. pp. 595–612 (2017)
28. Sekar, V., Egi, N., Ratnasamy, S., Reiter, M.K., Shi, G.: 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). pp. 323–336. USENIX, San Jose, CA (2012), https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/sekar
29. Shahbaz, M., Choi, S., Pfaff, B., Kim, C., Feamster, N., McKeown, N., Rexford, J.: Pisces: A programmable, protocol-independent software switch. In: Proceedings of the 2016 ACM SIGCOMM Conference (2016)
30. Shin, S., Gu, G.: 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. pp. 1–6. IEEE (2012)
31. Shin, S., Yegneswaran, V., Porras, P., Gu, G.: 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) (November 2013)
32. Snort: Network Intrusion Detection System, https://www.snort.org/
33. Sonchack, J., Aviv, A.J., Keller, E., Smith, J.M.: Enabling practical software-defined networking security applications with ofx (2016)
34. Suricata: An open source-based intrusion detection system (IDS), https://suricata-ids.org/
35. Tammana, P., Agarwal, R., Lee, M.: Simplifying datacenter network debugging with pathdump.
36. Yoon, C., Park, T., Lee, S., Kang, H., Shin, S., Zhang, Z.: Enabling security functions with sdn: A feasibility study. Computer Networks **85**, 19–35 (2015)
37. Zeng, H., Zhang, S., Ye, F., Jeyakumar, V., Ju, M., Liu, J., McKeown, N., Vahdat, A.: Libra: Divide and conquer to verify forwarding tables in huge networks. In: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). pp. 87–99. USENIX Association, Seattle, WA (Apr 2014), https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/zeng
38. Zilberman, N., Audzevich, Y., Covington, G.A., Moore, A.W.: Netfpga sume: Toward 100 gbps as research commodity. IEEE micro **34**(5), 32–41 (2014)