HEX Switch: Hardware-assisted security extensions of OpenFlow

Taejune Park
KAIST
taejune.park@kaist.ac.kr

Zhaoyan Xu
StackRox Inc.
z@stackrox.com

Seungwon Shin
KAIST
claude@kaist.ac.kr

ABSTRACT

Software-defined networking (SDN) and Network Function Virtualization (NFV) have inspired security researchers to devise new security applications for these new network technology. However, since SDN and NFV are basically faithful to operating a network, they only focus on providing features related to network control. Therefore, it is challenging to implement complex security functions such as packet payload inspection. Several studies have addressed this challenge through an SDN data plane extension, but there were problems with performance and control interfaces. In this paper, we introduce a new data plane architecture, HEX which leverages existing data plane architectures for SDN to enable network security applications in an SDN environment efficiently and effectively. HEX provides security services as a set of OpenFlow actions ensuring high performance and a function of handling multiple SDN actions with a simple control command. We implemented a DoS detector and Deep Packet Inspection (DPI) as the prototype features of HEX using the NetFPGA-1G-CML, and our evaluation results demonstrate that HEX can provide security services as a line-rate performance.

CCS CONCEPTS

• **Networks → Middle boxes / network appliances**; **Network security**;

KEYWORDS

SDN, Security, NetFPGA

ACM Reference Format:
Taejune Park, Zhaoyan Xu, and Seungwon Shin. 2018. HEX Switch: Hardware-assisted security extensions of OpenFlow. In *SecSoN'18: ACM SIGCOMM 2018 Workshop on Security in Softwarized Networks: Prospects and Challenges , August 24, 2018, Budapest, Hungary*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3229616.3229622

1 INTRODUCTION

The initial success of Software-Defined Networking (SDN) technologies in the networking community has motivated the security community to leverage them in devising novel security applications [11, 12, 18]. For example, FRESCO [13] has suggested a modular security service composition framework based on SDN to build new security functions easily, and a stateful firewall function has been instantiated as an SDN application [4]. Moreover some vendors have recently released SDN based network security applications as a product [3].

However, the abilities of those security applications are strictly restricted since current SDN and NFV proposals mainly consider network features not security. In the case of a network security application, investigating a packet payload (i.e., contents inside a payload) is an indispensable feature, but most network applications do not ask this ability since they mostly handle network headers instead of payload contents. As such, there are some fundamental requirement differences between network only applications and network security applications.

In this work, to enable network security applications in an SDN environment efficiently and effectively, we propose a new data plane architecture, HEX. HEX leverages existing data plane architectures for SDN (e.g., OpenFlow data plane specification) and adds three key functions that can help network administrators enable SDN network security applications easily. First, it makes an SDN application investigate packet payload just by issuing a simple command. With this feature, an SDN application can simply search payload to check whether it includes some known patterns, which is able to realize an IDS application. Second, it manages the counter information provided by SDN data plane (e.g., OpenFlow counter for each flow table) to make it more user-friendly. Third, it provides a function of handling multiple SDN actions with a simple control command (action clustering in Section 4). This method helps network administrators avoid managing many redundant flow rules and makes an SDN application implementation much simple.

When designing HEX, we also consider two critical requirements to make HEX more practical. First, all those new functions for security services should not ruin the concept of SDN. Specifically, our ideas should not make an SDN data plane architecture complicated. Thus, we try to minimally modify an SDN data plane components and optimally add new functions to an SDN data plane when implementing HEX. Moreover, those functions can be fully implemented in H/W without much difficulty to satisfy the high-performance requirement of a security application (e.g., intrusion detection function should handle 1 Gbps traffic). Second, all those new functions should be controlled by an SDN command or its extensions. Currently, SDN allows network administrators to control/manage network dynamically with a simple SDN application. We also inherit this philosophy of SDN, and thus we need to make all new functions be controlled by simple SDN commands (e.g., OpenFlow and its extensions).

To verify our ideas and check whether our design meets above two requirements, we have implemented a prototype system of HEX in H/W (based on NetFPGA [5]). Our initial implementation shows that our idea can be realized with minimal modifications of an existing SDN data plane architecture, and proves that our proposal can create high performance network security applications with

SecSoN'18, August 24, 2018, Budapest, Hungary

ACM ISBN 978-1-4503-5912-2/18/08…$15.00
https://doi.org/10.1145/3229616.3229622

simple commands (e.g., HEX can support the deep packet inspection function in a line-rate, 1 Gbps in our case).

2 PROBLEM STATEMENT

2.1 Security extension of SDN

Security services on SDN environment can be deployed as SDN applications in an SDN control plane (i.e., controller). However, since current SDN mainly focuses on network features over security, certain security services (e.g., difficulty in inspecting packet payloads) are unsuitable to be provided as SDN applications. Therefore, security services on SDN generally rely on virtualized network functions (VNFs) in middleboxes, but this could imply that all network traffic should take extra hops, resulting in significant performance degradation. Furthermore, an administrator should deploy extra flow rules to detour traffic to security services to satisfy operational security constraints.

To address the above challenges, prior studies have explored the possibility of extending an SDN architecture to provide security services on SDN itself. OFX [15] suggested the tap-interfaced security middlebox, and Mekky *et al.* [6] introduced app modules and app tables that enable customized packet handling. These studies were certainly valuable attempts, but they are structurally close to add-ons because their security functions are not consolidated into a data plane. That is, they are structurally equivalent to the middlebox-based VNF approach so they also suffer the same structural limitations of middleboxes: requiring extra hops and flow rules that cause performance degradation and the burden of management.

To demonstrate the full compatibility between the data plane and security functions without the inadvertent side effects, we proposed UNISAFE [9] which offloads security services into the packet processing sequences of a software switch to reduce the performance overhead and designs them as a set of OpenFlow actions, e.g., actions=`sec_dos(mbps=1000,...)`.

However, the security functions of these approaches are deployed and performed only at a data plane level. That is, once a security function is enforced by a controller, the security function continues to execute an indicated inspection, but the controller has difficulty in getting any information about the current security state. Therefore, dynamic security control through an SDN controller/application is challenging, which goes against the SDN's strengths of automation and programmability.

Furthermore, they only offered a simple statistical-based security feature such as bps counter, disregarding important features in security such as packet payload inspection. This is because previous studies have been implemented on a software basis, so it is difficult to cope with packet switching and complicated security functions at the same time with reliable performance. Although UNISAFE attempted packet payload inspection, the performance was sharply decreased, and it is challenging to deploy in a real network, i.e., the throughput was less than 100Mbps on 1Gbps network with the increase of latency.

2.2 Hardware-based implementation

In order to address these challenges, we extend UNISAFE to enable communication with the controller and design it as a new data plane architecture into hardware, called HEX. HEX can not only manage packet counters for the statistical-based security but also inspect packet payloads with line-rate speed. In addition, it can send an alert message to notify a current state to a controller through an OpenFlow channel when it detects any security violations, and the controller can operate security applications to handle the detected event.

In this paper, we present a design of a prototype HEX with a DoS detector (statistical-based security) and a Deep Packet inspector (payload inspection) using a NetFPGA [5] platform which is open-source hardware for rapid prototyping of network devices to prove our approach.

3 DESIGN

The main goal of HEX is to present a new data plane structure which directly provides security features to help a network administrator enable SDN security applications with ease. To achieve this goal, HEX provides security services expressed as an OpenFlow action notation, e.g., `actions=sec_dos(...), sec_dpi(...)`, and these *security actions* perform security inspection according to the match-action interface. Also, an SDN controller can enforce the security actions against network traffic to a HEX switch and receive an alert message when detecting security violation via a OpenFlow channel. For this work, we design security logic in hardware for its efficiency and performance.

3.1 HEX Security Processor

Fig. 1 illustrates the processing sequence of HEX switch. Similar to a typical OpenFlow switch, HEX consists of the packet parser (i.e., the packet preprocessor), the flow table controller and the action processor, and each module is sequentially executed. However, the main difference is that HEX has the specialized module to process the security action. This *HEX security processor* module is internally configured with a six-stage pipeline; 1) Buffering → 2) Read data storage → 3) Update Data storage → 4,5) Inspection sequence (2 stages) → 6) Policy Decision, and it runs before the action processor to inspect packets before being modulated by OpenFlow actions, e.g., *set_nw_src*.

Security Action: After matching the flow conditions in a flow table with incoming packets (i.e., parsing packets, indexing a flow table, updating flow statistics, and looking up flow entries corresponding to the packets), the flow table controller gets a *flow stats info* and a *action key*, and forwards them to the HEX security processor and the action processor. The action key is a data structure containing action flags that indicate which actions to be executed and its parameters such as output port numbers. The HEX security actions are also defined in the action key. For example, in case of a DoS detector action, an action key contains a DoS detector flag and bps threshold as its parameter (and also including a *policy* and *cluster ID* to be described in Section 4). When the HEX security processor module receives this key, the bandwidth usage is calculated for DoS inspection by referring to the flow stat info that comes together.

A security action is mainly processed with data storage and inspection logic; The data storage is implemented Block RAMs, and it stores necessary data for inspection. For example, a DoS detector

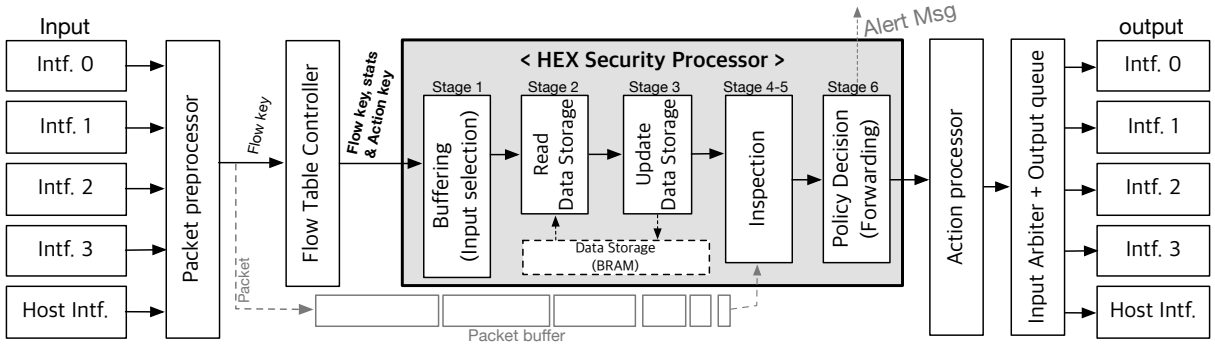

Figure 1: Processing sequence of HEX

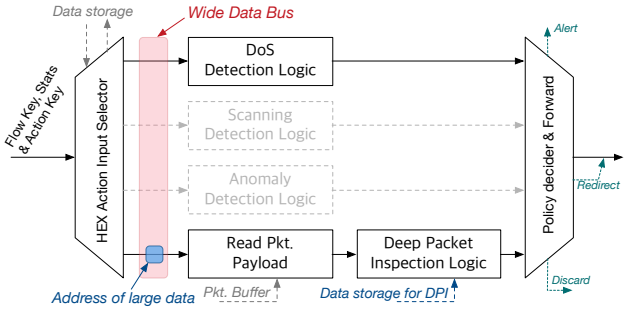

Figure 2: HEX security action processing

action stores accumulated bytes of incoming packets with its arrival time, and a Deep Packet Inspector (DPI) stores patterns to be used for a payload examination. These data are updated based on a *flow key* which is packet metadata used for indexing a flow table, and *flow stats info* which contains the statistical data (e.g., the count and bytes of packets) of each flow entry. The inspection logic conducts an actual packet inspection with the information in the data section. For instance, the processing logic for a DoS detector calculates the current bits per second (bps) using the accumulated bytes with the time information in its data storage.

When a security violation is detected after inspection, HEX handles a detected packet according to a *policy*, which is a special parameter all security actions have. The current version of HEX provides four policies; 1) `Neglect` ignores the violation and continues packet processing like a normal switch sequence, 2) *alert* sends an alert message including packet metadata, violating data (e.g., current bps), and a cluster ID (will be described in the next section) to a controller, 3) *discard* terminates the packet processing sequence and drops the detected packet, and 4) *redirect* forwards packets to alternative destinations (e.g., honeypot) instead of the original destination specified in a flow rule.

Action processing: Figure 2 describes the detail of the security processing in the inspection stages After receiving an action key which contains flags containing security actions, flow key, and flow stats, HEX discriminates what security actions will be performed. Then HEX reads and updates related data of selected actions in data storage, and the updated data are transmitted simultaneously as reference data for inspection to all inspection logic through a wide data bus. That is, all inspection logic is executed in parallel, which allows us to provide multiple security services at a much faster than sequential processing.

Some security actions which examine a packet payload, such as DPI or Application-layer firewall, have a slightly different process to read a packet payload. Before entering inspection logic, they have a process to read a packet from the packet buffer, then the and reference data are transmitted to the inspection logic together. If a reference data size that is too large to contain on the data bus, the inspection logic can access the data storage directly. In this case, instead of full reference data, an address of it on the data storage is transferred by the data bus so that it can be accessed by the inspection logic. For example, when a DPI action should check a large number of patterns, the data bus delivers an address of the patterns on the data storage instead of whole patterns, and the DPI inspection logic accesses the data storage additionally during the inspection, as shown in Figure 2.

After inspection, if any security violations are detected, HEX handles a detected packet in accordance with a policy by combining all results. If policies are conflicted between security actions, the packet will be processed following a higher priority policy; the priority of policies is determined as Redirect → Discard → Alert → Neglect (High to low order). The policy is implemented by modifying an output port list in the action key which will be forwarded the next module (i.e., action processor). For instance, the redirect policy sets the output port list to an alternative port number, and the drop policy sets the output port list to not forward anywhere.

3.2 Communication with a host

To forward a generated alert message to a controller and deploy a security service from a controller, HEX switch relies on a host software as depicted in Figure 3. The HEX switch hardware and the host software are connected through a device driver, and the device driver enables the hardware and software to communicate each other by reading and writing registers exposed to the outside in the hardware part.

Transferring an alert message: When an alert message occurs in the HEX security processor, this alert message is stored in the HEX message registers at first. If a value change is detected in the HEX message registers, the device driver delivers the values of the HEX registers to the HEX message handler in the software area. Then, the HEX message handler builds a message and sends it to the controller via the OpenFlow channel.

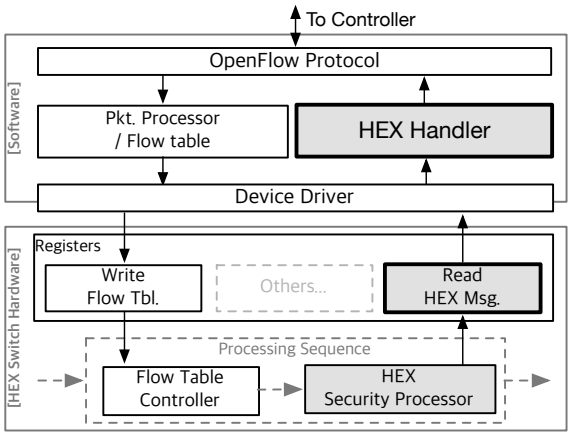

Figure 3: Message handling of HEX

Deploying security actions: Since security functions are designed as parts of OpenFlow actions, a controller can easily utilize security actions to apply them to network traffic. By simply appending security actions into a `Flow_Mod` message like common OpenFlow actions, an administrator can enforce security services to a specific network flow. The Flow_Mod message is delivered in the flow table through the device driver with flow table registers, and the HEX switch will execute security actions defined in the action key of the installed flow rule.

By combining the above two points, we can build a HEX application on a controller like Algorithm 1; As shown in the `Install_Securities` function, HEX is compatible with a normal OpenFlow protocol, we can use the HEX actions like common OpenFlow actions. Also, as shown in the `HANDLER_HEX_MSG` function, a HEX application can listen an alert message from HEX. After receiving the alert message, the application can respond with appropriate reactions to abnormal behaviors, e.g., deploying collecting attack behavior or blocking the flow rules.

4 ACTION CLUSTERING

We devised *action clustering* in UNISAFE that is a novel technique which merges the HEX security actions of multiple flow rules into a few synthetic rules. It allows HEX security actions between different flow rules to be performed as if they are executed for a single rule, hence it compresses a complicated flow table in simple.

Algorithm 1: HEX security application pseudo code

```
Procedure Install_Securities:
    msg ← openflow.flow_mod()
    msg.match.in_port ← 1
    msg.actions[0] ←
        hex.sec_dos(mbps=1000,id=10,policy=redirect:2)
    msg.actions[1] ←
        hex.sec_dpi(rule=pattern,id=20,policy=alert)
    send_to_controller(msg)

Procedure HANDLER_HEX_MSG (alert):
    packet ← event.packet // This is the parsed packet data
    print "in_port: alert.in_port"
    print "alert_reason: alert.reason"
    print "cluster_id: alert.cluster_id"
    print "value: alert.value"
```

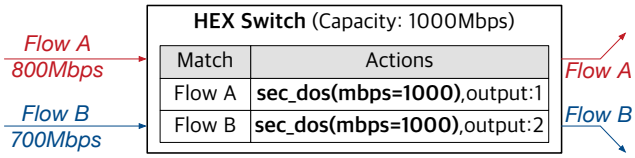

Figure 4: Challenge in security action enforcement between two flows

4.1 Introduction of Action Clustering

We present action clustering with a challenge of flow-level security actions deployment. In the Figure 4, there is a HEX switch and its bandwidth capacity is 1000 Mbps. Let us assume that an administrator wants to detect a DoS attack for this network, thus the administrator enforces a DoS detector action which monitors the 1000 Mbps threshold in two different flows `Flow A` and `Flow B`. In this situation, when Flow A and B pass the switch at 800 and 700 Mbps, the total incoming bandwidth evidently exceeds 1000 Mbps. Unfortunately, the DoS detectors never trigger an alert because each flow itself does not exceed 1000 Mbps.

To relieve this problem, the administrator may conservatively try to configure the DoS detectors with a 500 Mbps threshold. However, with this approach, if Flow A uses 800 Mbps of bandwidth but Flow B does not have any traffic, the DoS detector would trigger an alert against Flow A although 200 Mbps of bandwidth is still available. Because this problem fundamentally occurs due to per-flow state isolation, to address this challenge correctly, the administrator should install extra flow rules to aggregate flow rules such as multi-tables or group tables, e.g., Table 0: aggregating Flow A and B → Table 1: DoS inspection for the aggregated flow → Table 2: separating and forwarding Flow A and B.

However, the multi-tables or group tables approach inevitably escalates the complexity of the flow table, which also introduces difficulties in terms of network service management. To address this challenge, action clustering is utilized; All security actions have a *cluster ID* as an additional parameter, and same security actions that use same cluster IDs are considered to be in the same cluster. The security actions that belong to the same cluster works as if they were a integrated single action across different flow rules. Therefore, flow rules of Figure 4 can be written as

```
Flow_A: actions=sec_dos(mbps=1000,id=10),output:1
Flow_B: actions=sec_dos(mbps=1000,id=10),output:2
```

The Flow A and B rules have the same cluster ID 10 for the DoS detector action. The DoS detectors in the different flow rules run like a single action for both flows, thus they can accurately detect a total of 1500 Mbps of traffic for both flows at 800 and 700 Mbps.

4.2 Design of Action Clustering in HEX

Action clustering is achieved by sharing the same reference data in the data storage as described in Figure 5. Based on the type of security actions and the cluster IDs, the data storage builds the clustering map per security action and maintains the aggregate data per cluster ID. That is, to read data storage, action flags which

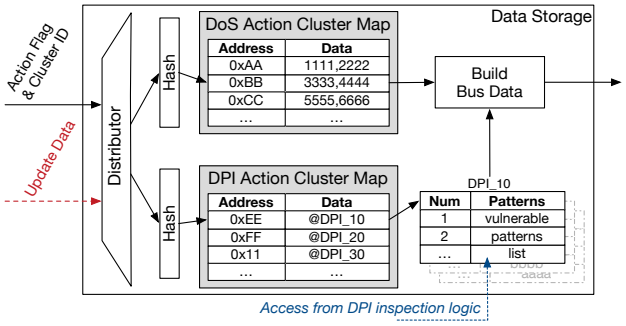

Figure 5: Data structure of Data storage

are selected by the HEX input selector are transferred to the data storage with its cluster IDs, and the cluster ID is used as a hash key of the memory address. Therefore, the security actions with the same cluster ID refer to the same reference data. After reading the reference data, the data storage collects and returns it to the data bus. In terms of updating the data, the data to be updated are sent together with the action flags with its cluster IDs. Then, the reference data of actions involving in the same cluster are accumulated together in the same address area even if they are different flows.

Some security actions that have a reference data size that is too large to contain on the data bus may have a two-layered memory structure to support action clustering. For example, DPI requires a two-dimensional memory area to store patterns (commonly string structure). Hence, the cluster map manages memory areas for storing rules, and a cluster ID refers to this location as shown in Figure 5. When reading the data storage, the location of this memory is transferred to the data bus, and DPI accesses this data directly to get a pattern list as mentioned in the previous section.

5 IMPLEMENTATION

We have implemented a HEX prototype using the NetFPGA-1G-CML [2] based on the NetFPGA 10G OpenFlow Switch [7, 17]. We have migrated the 10G OpenFlow switch to the our 1G-CML board and deployed a HEX security processor by modulating its action and flow processing modules. We have also implemented the HEX software parts (i.e., the device driver and HEX handler) applying the set of reference software included in the reference_nic_nf1_cml project.

The current version of HEX provides the DoS detector and Deep Packet Inspection (DPI) action and supports 1024 and 4 clusters, respectively. It is worth to note that our design principles will also apply to various security services such as a scanning detector or anomaly detector, and support more cluster spaces. These works are currently under development and remain as the future work.

6 EVALUATION

The evaluation environment consists of two hosts (i.e., h1 and h2), and a switch PC. All machines are connected to the NetFPGA-1G-CML board, the host PCs run the NetFPGA as a reference NIC, and the switch Pc runs as HEX as depicted in Figure 6. The h1 machine has Intel i7-4790@3.6GHz CPU, and others have Intel i3-7100@3.90GHz CPU.

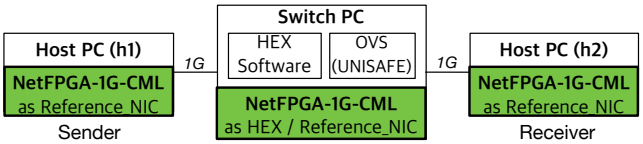

Figure 6: Evaluation Environment

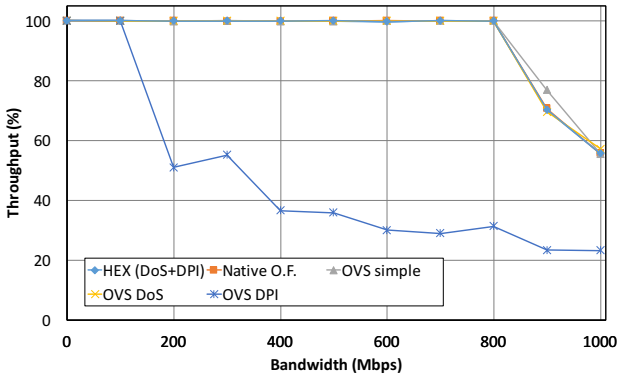

Figure 7: Throughput

We measured throughput and latency of our HEX switch running the DoS detector and DPI (100 rules) concurrently, and a native NetFPGA-1G-CML OpenFlow switch to compare performance degradation. Furthermore, in order to compare the performance improvement compared to the software solution, we also implemented a software-based solution (i.e., UNISAFE [9]) with DoS detector and DPI (100 rules) using Open vSwitch (OVS) [16] with a reference NIC in the Switch PC, and measured their throughput and latency.

6.1 Throughput Measurement

We first evaluated throughput by sending 1M random packets from h1 to h2 with different transmission rates. As shown in Figure 7, the throughput of HEX is indistinguishable from the native OpenFlow switch. Even after 800 Mbps, where the performance degradation begins to take off in earnest, HEX has performance degradation at the same rate as the native OpenFlow switch, and this implies the performance degradation of HEX is negligible.

These results also indicate that HEX offers a compelling performance improvement over a software-based solution; The simple OVS without any security services shows nearly the same throughput as HEX and the native OpenFlow switch, and also the OVS with the DoS action is almost no throughput degradation. However, in case of the DPI on OVS, we can see that there is a noticeable performance degradation. Even at 200 Mbps points, the throughput is reduced to 50%, and it can handle only 30 40% compared to the simple OVS or HEX.

6.2 Latency Measurement

We have also measured latency of the HEX switch by measuring a RTT time of ping. Figure 8 illustrates its CDF graph. In most cases, latency of HEX approaches that of latency of the native OpenFlow switch; While HEX provides security functions, 99% of packets are processed in less than 0.25 ms, which is the same as the native

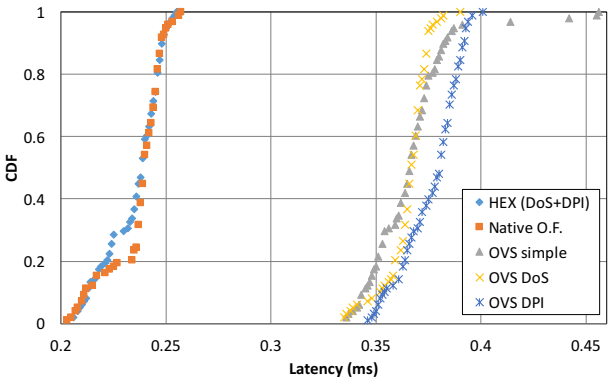

Figure 8: Latency

OpenFlow switch. This result clearly indicates that HEX has no overhead in terms of latency.

The latency of HEX is also remarkable when we compare HEX with the software solution. Even if OVS does not run anything (i.e., OVS simple), the latency is increased by approximately two times when compared to HEX. Also, the latency of the DPI on OVS is increased slightly more than the simple OVS.

7 DISCUSSION

Our evaluation results show that the previous software solution (i.e., UNISAFE) provides a simple security feature (i.e., DoS) with little performance degradation but has the overhead for operating a heavy security service (i.e., DPI) due to the architectural limitations of the software. On the other hand, HEX provides security services without performance degradation, and we have positive expectations in the deployment of HEX in real networks. This implies that the approach to designing security services in the data plane is an efficacious method depending on the implementation manner.

In addition, HEX allows a network administrator to manage the security actions using HEX applications, and action clustering helps to reduce the number of flow rules in the software switch. We believe that deploying security services on a network with HEX will help relieve configuration, management and debugging workloads.

In the case of when HEX cannot replace all security devices (or software) in a network, we expect that HEX can cooperate with them in various ways according to a network condition. For example, rather than carry all security checks on middleboxes or NFV services, they simply process packets filtered by HEX, instead of all packets, using a `redirect` policy to redirect packets. Furthermore, we believe that this cooperation can be also applied to compose other network security systems such as honeypot, reflectornets, and Moving Target Defense(MTD) because it provides a way to control packets according to their conditions.

Limitations First of all, deploying new features in HEX is challenging compared to software due to the low programmability of the hardware. Instead, by providing a simple conditional statement such as *if* in the future, we expect to provide some programmability per flow level.

Because all security actions run simultaneously, HEX can provide multiple services for a flow, but cannot set the priority between each service (i.e., the order of service chain). In addition, even if an administrator deploys the same action multiple times in a flow, only one action can be performed. For example, to install a DoS detection rule for 'alert at 100mbps, drop at 1000mbps', it logically could be written as *actions=sec_dos(mbps=100, policy=alert), sec_dos(mbps=1000, policy=discard)*, but it does not work in HEX. This problem is caused by the unidirectional processing of the current HEX. We will introduce the ability to perform service orders and iterating operations in the future.

Furthermore, due to limitations in memory capacity, HEX cannot allocate a sufficient cluster space and volume of DPI rules. We expect that this limitation could be addressed by replacing memory from BRAM to SDRAM.

8 RELATED WORK

OFX [15] is an OpenFlow extension framework which enables an OFX application to be loaded into a switch at a runtime. Mekky *et al.* [6] have suggested an extended SDN architecture to handle packets through modified flow tables on a switch called app tables. Both works are similar to HEX in terms of processing packets on a switch. However, they use additional control interfaces(i.e., API) or rules to manage imported network services, and due to architectural structure, the performance is inferior to HEX. QoSE [8] proposed a data plane module for providing security features by applying distributed NFV. However, the structure that goes through multiple hops in QoSE is a disadvantage in performance. Avant-guard[14] proposes a secure OpenFlow switch architecture that provides the connection migration and actuating triggers to enhance the scalability and responsiveness of OpenFlow switches. However, it only cares about TCP-based flooding attacks, while HEX can not only detect the flooding attack but also provide more security services in the future. IX [1] and zygos [10] proposed the operating system for high throughput and low latency. These studies allow applications to handle packets rapidly. We expect that these studies can be fully exploited for internal memory processing for HEX security functions.

9 CONCLUSION

We presented the design of a new data plane structure called HEX that embeds security functions. HEX simplifies deploying security services and provides security services without overheads. In addition, it supports action clustering thereby decreasing the number of flow rules and reducing the complexity of a flow table. We implemented the prototype HEX with two security services into NetFPGA, and our evaluation demonstrates that HEX delivers security services with negligible overhead. We believe that this result is quite encouraging, and will continue research to enable HEX to operate at even higher speeds with more security services.

Acknowledgement

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2016-0-00078, Cloud based Security Intelligence Technology Development for the Customized Security Service Provisioning)

REFERENCES

[1] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2017. The IX operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Transactions on Computer Systems (TOCS)* 34, 4 (2017), 11.

[2] NetFPGA Group. 2014. NetFPGA-1G-CML. (2014). https://netfpga.org/site/#/systems/2netfpga-1g-cml/details/

[3] HP. 2013. HP SDN App Store. (2013). https://marketplace.saas.hpe.com/sdn

[4] Hongxin Hu, Wonkyu Han, Gail-Joon Ahn, and Ziming Zhao. 2014. FLOWGUARD: building robust firewalls for software-defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 97–102.

[5] John W Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. 2007. NetFPGA–an open platform for gigabit-rate network switching and routing. In *Microelectronic Systems Education, 2007. MSE'07. IEEE International Conference on*. IEEE, 160–161.

[6] Hesham Mekky, Fang Hao, Sarit Mukherjee, Zhi-Li Zhang, and T.V. Lakshman. 2014. Application-aware Data Plane Processing in SDN. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN '14)*. ACM, New York, NY, USA, 13–18. https://doi.org/10.1145/2620728.2620735

[7] NetFPGA GitHub Organization. 2012. NetFPGA 10G OpenFlow Switch. (2012). https://github.com/NetFPGA/NetFPGA-public/wiki/NetFPGA-10G-OpenFlow-Switch

[8] Taejune Park, Yeonkeun Kim, Jaehyun Park, Hyunmin Suh, Byeongdo Hong, and Seungwon Shin. 2016. QoSE: Quality of SEcurity a network security framework with distributed NFV. In *Communications (ICC), 2016 IEEE International Conference on*. IEEE, 1–6.

[9] Taejune Park, Yeonkeun Kim, and Seungwon Shin. 2016. UNISAFE: A union of security actions for software switches. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*. ACM, 13–18.

[10] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 325–341.

[11] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. 2013. SIMPLE-fying middlebox policy enforcement using SDN. In *ACM SIGCOMM computer communication review*, Vol. 43. ACM, 27–38.

[12] Seungwon Shin and Guofei Gu. 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, 1–6.

[13] Seungwon Shin, Phillip A Porras, Vinod Yegneswaran, Martin W Fong, Guofei Gu, and Mabry Tyson. 2013. FRESCO: Modular Composable Security Services for Software-Defined Networks.. In *NDSS*.

[14] Seungwon Shin, Vinod Yegneswaran, Phil Porras, and Guofei Gu. 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)*.

[15] John Sonchack, Adam J Aviv, Eric Keller, and Jonathan M Smith. 2016. Enabling Practical Software-defined Networking Security Applications with OFX.

[16] Open vSwitch. 2011. An Open Virtual Switch. (2011). http://openvswitch.org/

[17] Tatsuya Yabe. 2011. OpenFlow implementation on NetFPGA-10G Design Document. (2011).

[18] Changhoon Yoon, Taejune Park, Seungsoo Lee, Heedo Kang, Seungwon Shin, and Zonghua Zhang. 2015. Enabling security functions with SDN: A feasibility study. *Computer Networks* 85 (2015), 19–35.