Computer Networks 85 (2015) 19–35

Contents lists available at ScienceDirect

Computer Networks

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

Enabling security functions with SDN: A feasibility study

Changhoon Yoon [a,*], Taejune Park [a], Seungsoo Lee [a], Heedo Kang [a], Seungwon Shin [a], Zonghua Zhang [b]

[a] KAIST, Graduate School of Information Security, 291 Daehak-ro, Yuseong-gu, Daejeon, South Korea
[b] Institute Mines-Tlcom/TELECOM Lille of France, Department of Computer Science and Network, Rue Guglielmo Marconi, 59650 Villeneuve-d'Ascq, France

ARTICLE INFO

Article history:
Received 17 September 2014
Revised 10 April 2015
Accepted 5 May 2015
Available online 21 May 2015

Keywords:
Network security
Software-defined networking security
SDN security

ABSTRACT

Software-defined networking (SDN) is being strongly considered as the next promising networking platform, and studies regarding SDN have been actively conducted accordingly. However, the security of SDN remains undefined and unknown when considering the enhancement of network security in SDN. In this paper, we verify whether SDN can enhance network security. Specifically, the idea of enabling security functions with diverse SDN features is explored thoroughly. In order to elucidate the feasibility of SDN-based security functions, we implement four types of security functions with SDN in Floodlight applications: (i) in-line mode security functions (e.g. firewalls and IPS), (ii) passive mode security functions (e.g. IDS), (iii) network anomaly detection functions (e.g. scan and DDoS detector), and (iv) advanced security functions (e.g. stateful firewall and reflector networks). Furthermore, we focus on discovering issues that might arise throughout the implementation of SDN-based security applications and discuss how these issues can be addressed. In order to appropriately prove the feasibility of the SDN-based security applications, we evaluate our Floodlight applications in real testbeds that consist of SDN-enabled switches and a number of physical hosts.

1. Introduction

Attracting significant attention from both academia and industry, software-defined networking (SDN) has quickly emerged as a new promising technology for future networks. SDN separates the control plane from the data plane, and thus it enables the easy addition of new, creative, and powerful network functions/protocols. In academia, a number of research ideas based on SDN/OpenFlow have been proposed [1–8] since the publication of OpenFlow [9], which is a key component in realizing the SDN concept. In industry, SDN is widely considered to be the new paradigm for future networks. Many companies are deploying or planning to deploy this technology in their system in order to strengthen their network architectures, reduce operational costs, and enable new network applications/functions. In 2011, companies including Google, Juniper, Facebook, and Microsoft have formed a specific organization, i.e. the Open Networking Foundation [10], in order to accelerate the delivery and use of SDN through promoting OpenFlow. In general, SDN is considered to be a critical information technology trend over the next five years [11,12]. By 2016, the estimated knowledge discovery investment for SDN is estimated to be approximately US$2 billion [13]. Likewise, SDN is widely employed and is being used in real world applications by pioneers around the globe.

However, there is one area of SDN that requires further development: *security*. As previously stated, SDN is considered to be a significant future network technology, and it changes the current network architectures and services.

* Corresponding author. Tel.: +82 1053517101.
E-mail addresses: chyoon87@kaist.ac.kr (C. Yoon), taejune.park@kaist.ac.kr (T. Park), lss365@kaist.ac.kr (S. Lee), kangheedo@kaist.ac.kr (H. Kang), claude@kaist.ac.kr (S. Shin), zonghua.zhang@telecom-lille.fr (Z. Zhang).

http://dx.doi.org/10.1016/j.comnet.2015.05.005
1389-1286/

Therefore, *have security researchers (or practitioners) used the SDN technology to enhance or replace the current security functions?* In our survey, only a handful of researchers have used the SDN technology to enhance or replace the current security functions. For example, methods to detect network anomalies (e.g. denial of service and network scans) [1,14] or to enable firewall functionality [15] with SDN have been proposed, and a framework that easily implements security functions has been recently investigated [8]. This narrow investigation of SDN is not limited to academia, but is also found in industry. For example, vArmour announced that they would release a virtual firewall that can dynamically control network flows [16], and we were not able to find any other noticeable examples of commercialized SDN security service products. Only the few examples of SDN implementation demonstrate that it could be used for security purposes (i.e. specifically, realizing network security functions).

While many possibilities for designing security functions with SDN can also be located, observing the real world applications of SDN-based security functions remains challenging. Moreover, most existing studies have only presented research prototypes, not practical implementations. Another question that should be addressed is: *Can SDN technology help in implementing decent network security functions or devising advanced security functions?* However, to date, there has not been significant research undertaken to understand this problem.

In order to answer this question, we implemented several network security functions (e.g. firewalls, network anomaly detectors, etc.) using SDN technology in Floodlight [17] applications. Floodlight is an enterprise-class, open-source, Java-based OpenFlow controller, which is also the core of a commercial controller product from Big Switch Networks [18]. Then, we analyzed them in various scenarios. Based on our experiences while implementing these security functions, we garnered diverse insights that will encourage others to design different network security functions. From this work, both researchers and practitioners can understand how they could implement security functions without adding third party devices. Through the example cases presented here, we believe that researchers and practitioners could gain inspiration to create their network security functions with SDN. In addition, we investigate the diverse performance metrics of the implemented functions in order to understand if they are applicable in real world network. In order to understand this, we explored the functions in a test-bed that represents a real network environment. These analysis results can elucidate the following problem: *Can we use SDN-based security functions in real world settings?*

The contributions of this paper are summarized as follows.

- To date, our work is the first trial that examines real and practical issues regarding the implementation of diverse network security functions with SDN.
- We implement four different types of network security functions with SDN that can be used in real life cases; we also describe their design issues in detail. We believe that our experience of designing these security functions can assist other researchers and practitioners to devise more practical network security functions and potentially commercial products. In addition, our codes are in the public domain in order to encourage other people to develop better and more practical security functions.
- We provide insights and discussions based on our experience. We believe that these insights and discussions will assist others to reduce mistakes in developing practical SDN-based security functions. In addition, others can design more intelligent security functions based on our case study. For example, in one of our experiments, we demonstrate that we can implement a ReflectorNet (Section 5), which was once a difficult task, through writing 400 lines of code with the aid of SDN technology. It is our hope that anyone can realize state-of-the-art security functions through our insights and findings.

In Section 2, we introduce the conceptual design of a simple firewall function that could be enabled using SDN technology and the possible challenges or issues that might arise when it was deployed on a real network in order to illustrate our problem domain. We present the research questions in this section as well, and we give the answers to the questions in the rest of the sections by implementing four types of security functions: (i) in-line mode (Section 3), (ii) passive mode (Section 4), (iii) network anomaly detection (Section 5), and (iv) advanced security functions (Section 6). We believe that these four types of security functions represent the diversity of currently available security functions. Thus, we expect that the investigation of these four functions will reveal most possible issues that might arise in implementing and deploying SDN security applications. Section 7 attempts to verify if the four types of SDN-based security function implementations are feasible to be deployed to real networks by evaluating each application that we implement in this paper.

2. Motivating example and research question

In this section, we introduce an example SDN application that serves as a simple network security function to describe the problem that is addressed in this paper.

2.1. Motivating example

As the SDN technology becomes more widely deployed and accepted, a number of interesting network applications for the SDN platform have been devised, such as load balancing [19], WAN management [20,21], and network monitoring [22] applications. Likewise, both researchers and practitioners have interests in devising security functions (specifically, network security applications) using SDN. For example, a technique that detects denial of service (DoS) attacks with SDN functions has been proposed [1], scan detectors have been implemented with SDN [14], and a novel framework for developing security applications has been developed [8].

Most of the modern networks employ dedicated middleboxes for security purposes; however, such devices are costly, bulky, and inflexible. If an SDN application could replace the hardware-based network security appliances, it would enable incredible benefits. Then, *how can we devise network security functions with SDN?* In order to provide a foundational answer to this question, we present a naive SDN-based firewall application (A more profound SDN-based firewall such

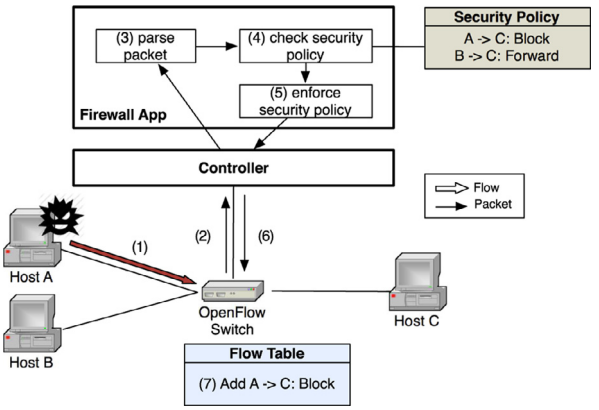


Fig. 1. Conceptual firewall function implementation.

as FlowGuard [15] exists as well). With our firewall example, we illustrate how the firewall function could be enabled in an SDN platform. The firewall application, as depicted in Fig. 1, issues flow rules in order to enforce an access control list (ACL) to deny malicious trials.

In order to be more specific, a network operator may specify a firewall policy or security policy. In Fig. 1, the operator has set the firewall policy to block the flow that is traveling from Host A (a known malicious host) to Host C. On the data plane, when (1) Host A attempts to send a network packet to Host C, the OpenFlow switch checks its flow table if a flow rule entry to handle the packet exists. If no match is found, (2) the packet is sent to the controller. Then, the firewall application deployed in the controller receives the packet as well as the other deployed applications. The firewall application (3) parses the packet and (4) checks if it matches any rule specified in the firewall policy. Since the packet matches the rule "A→C: Block", (5) the application enforces the corresponding policy through instructing the controller to block all packets that fall into the same flow. Then, the controller (6) issues a flow rule to block the flow; finally, (7) the rule is installed in the flow table of the switch. Therefore, traffic that travels from Host A to Host C is dropped at the switch as long as the rule remains in the flow table.

As this example illustrates, a firewall feature (even a distributed firewall) can be enabled without adding third party devices through enabling a firewall function at each network device. This straightforward implementation scheme appears to be plausible; however, the firewall application might not be as feasible as expected if it was deployed in real networks. For example, it might issue excessive numbers of flow rules and consume all flow table entries, which ultimately causes collateral damage to the entire network. In addition, the firewall application might consume a significant amount of resources, and consequently, incur overall performance degradation to the network.

Likewise, regardless of how complex the security application is, various problems may be faced in the SDN environment. An SDN application for network security services could fail in real network settings or even face serious challenges due to unforeseen constraints of the SDN environment. To date, most existing studies have discussed the security issues surrounding SDN [1,8,14,15]. In addition, they have only provided some research prototypes, and their performance in real networks has not been verified. Prior to the extensive implementation or deployment of SDN-based network security applications, their performance in real networks (or testbeds with real devices) must be verified, and any notable issues/constraints that might appear during the implementation or deployment of the application must be carefully examined, analyzed, and reviewed.

2.2. Research question

The firewall example presented in Fig. 1 implies that employing SDN technology in developing security functions will face some challenging issues. The primary goal of this paper is to examine whether SDN technology can be leveraged in implementing practical, effective, and efficient network security functions. In order to achieve this goal, the following key research questions must be investigated:

1. Can SDN-based security applications potentially replace existing security appliances?
2. If possible, how can SDN-based security applications be devised appropriately?
3. If not possible, what are the issues and how can they be addressed?

If further information regarding or answers to these questions can be obtained, the goal is naturally achieved; thus, the remainder of this paper focuses on discovering the answers to each question.

3. In-line mode security applications

The prevalent in-line mode security functions were implemented in Floodlight applications, and detailed descriptions of their design and operation are provided in the following. In this section, we describe how each in-line mode security function would behave on an SDN platform, then introduce how each function could be implemented in Floodlight application, and discuss about what are the advantages and disadvantages of implementing this type of security functions using SDN.

3.1. Firewall application

A Floodlight application with firewall functionality is packaged with the Floodlight controller distribution. Although it is under development (April 2014), the Floodlight application used here is capable of performing basic firewall tasks such as enforcing Access Control List (ACL) on OpenFlow enabled switches (depicted in Fig. 1).

Since only limited and outdated information were provided in the Floodlight documentation, we have analyzed the source code of the firewall application in order to completely understand its operation. The operation of the Floodlight firewall application is predominantly the same as the example firewall operation introduced in Section 2 (Fig. 1).

The design of the firewall application is depicted in Fig. 2. Upon initialization of the firewall, it reads the firewall rules from the persistent database storage located in the controller (via the *IStorageSourceService* interface). The rules are maintained in a sorted array in the memory, and they are sorted in

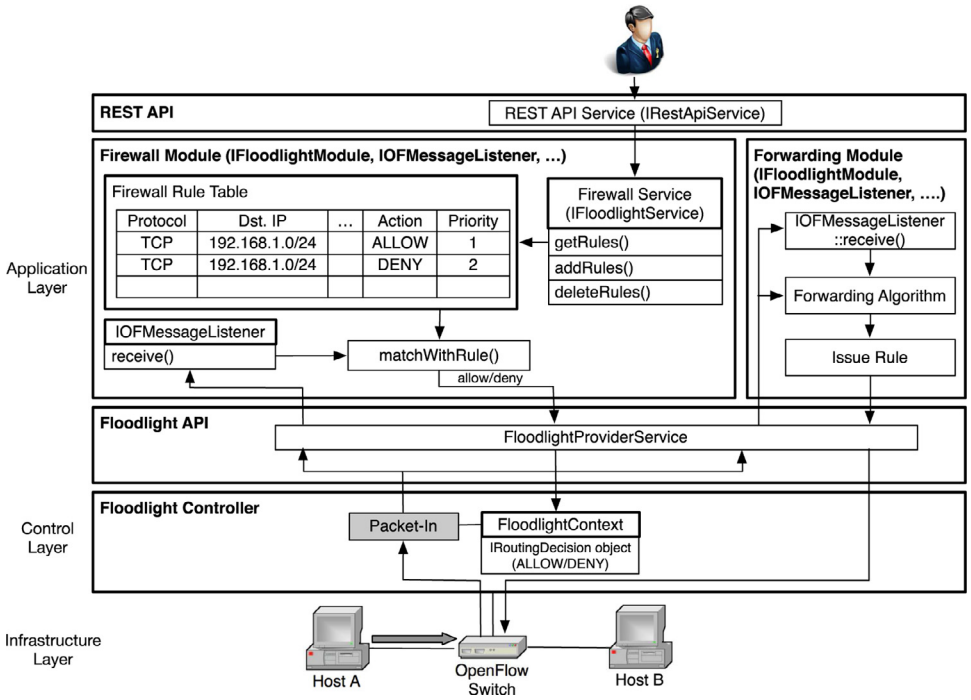


Fig. 2. Firewall application implementation.

decreasing order of assigned priority. Then, the firewall participates in the OpenFlow (OF) message processing pipeline as an *IOFMessageListener* instance and captures the *Packet-In* (the first packet of network flow) messages.

For each incoming *Packet-In* message, the firewall compares the header fields against each rule in the sorted list sequentially from the highest priority. If the firewall finds a matched entry, it stores the matching rules action (either ALLOW or DENY a flow) in an *IRoutingDecision* object; this *IRoutingDecision* object is added to the *FloodlightContext* object that is created and dedicated to each *Packet-In* message. This *FloodlightContext* object is also shared by other applications in the pipeline and thus it allows the Forwarding application (the packet forwarding application included in the Floodlight distribution) to recognize the firewall decision. The firewall imposes a prohibitive policy through denying all traffic by default; furthermore, it aims to match every single packet against the firewall rules. Hence, the Forwarding application uses *Packet-Out* messages to forward each packet. It forwards a packet through sending a *Packet-Out* message with an appropriate action for an ALLOW decision, while it drops a packet through sending a *Packet-Out* message without specifying an Output action for a DROP decision.

3.2. NIPS application

A network intrusion prevention system (NIPS) function is one of the most prevalent network security functions: it actively detects and blocks any intrusion attempts on the network.

Despite the rich benefits of SDN, the separation of the control plane and data plane can become inefficient in some specific cases. Particularly in an SDN network using OpenFlow, the delivery of full packet information from the data plane to the control plane is not supported (only packet header information can be delivered to the control plane via *Packet-In* message). Meanwhile, NIPS function implemented in the SDN application needs to perform a full packet inspection against every single ongoing network packet on the data plane, and hence enabling the full packet delivery to the control plane is one challenge that needs to be solved. Furthermore, NIPS must be in-line with the network traffic in order to effectively block intrusion attempts, and satisfying this in-line requirement might also be a challenge to enabling NIPS functions on SDN. These challenges must be addressed in order to enable NIPS functions on SDN platforms.

Conceptual design: We present an example scenario that resolves the challenges stated above and thereby enables NIPS functions on SDN platforms. The scenario involves three hosts (A, B, and C) connected to an OpenFlow enabled switch as illustrated in Fig. 3.

In this scenario, we assume that the intrusion prevention service is provided to the traffic that flows from Host A to Host C. The NIPS application initiates as (1) Host A sends data traffic to Host C. When the traffic reaches the OF-enabled switch, (2) the first packet of the flow is sent to the controller and (3) it is passed to the NIPS. Then, the packet is analyzed in order to determine if the entire flow should be forwarded to the NIPS. (4) The NIPS notifies the controller that the flow should be forwarded to the NIPS, and (5) the controller issues the corresponding flow rule to instruct the switch to do so. (6) The flow entry of "Add A→C: Forward to NIPS" is added to the flow table of the switch; hence, (7) the flow is forwarded to the NIPS. The NIPS captures all packets forwarded by the switch, and (8) the packets undergo deep packet inspection, and any malicious packets are dropped at this point. (9) The

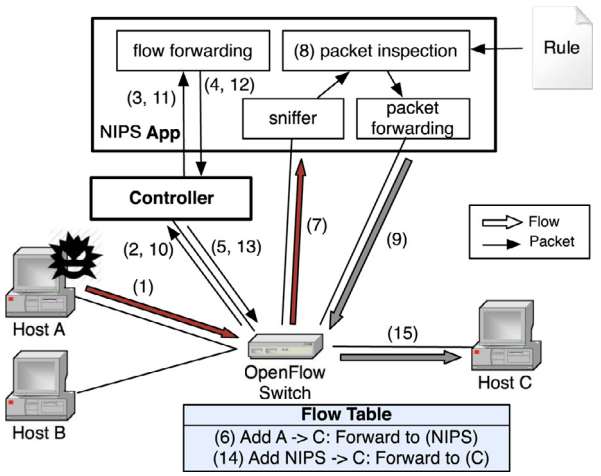


Fig. 3. Conceptual NIPS function implementation.

sanitized flow is returned to the switch, and (10) the switch queries the controller through sending the first packet of this newly created flow coming from the NIPS. Then, (11–13) the NIPS notifies the controller that the flow should be forwarded to its original destination because the flow has already been to the NIPS. Next, (14) the flow entry of "Add NIPS→C: Forward to C" is added to the flow table, and finally (15) the flow arrives at its final destination.

Floodlight implementation: The design of our Floodlight NIPS application is depicted in Fig. 4. The application consists of three major components: the *Forwarding*, *PacketHandler*, and *PacketInspection* modules. The *Forwarding* module is designed to operate in a reactive manner. Upon the arrival of a *Packet-In* message, the *Forwarding* module captures it using the *IOFMessageListener* and builds a flow rule that instructs the OF-enabled switch to forward the flow to the NIPS application. However, such forwarding is not possible because the NIPS application is not a valid network node visible to the switch. In order to turn the NIPS application into a legitimate network node, another dedicated network interface was deployed between the control plane and data plane as depicted in Fig. 4.

In order to instruct the network traffic to travel from the source to the destination via the NIPS application, at least two flow rules must be installed on an OF-enabled switch. One flow rule forwards the network traffic that has not gone through the NIPS application (or suspicious traffic) to the NIPS, and another flow rule forwards the traffic that has gone through the NIPS (or sanitized traffic) to the destination. With the OpenFlow's matching capability in the ingress port, it is possible to efficiently determine if the traffic is sanitized or not. In order to exploit this capability, we supplied the ingress port number, which was dedicated for the NIPS use, to the *Forwarding* module. Through specifying the ingress port number in the flow rule, it is possible to enforce different forwarding policies for the sanitized traffic and suspicious traffic. The *Forwarding* module forwards the sanitized traffic to the destination, while it forwards suspicious traffic to the NIPS, thus placing the NIPS in-line with the traffic.

Then, the network traffic forwarded to the NIPS application is captured by the *PacketHandler* module, and each captured network packet undergoes the packet inspection process. We implemented the *PacketInspection* module to compare the network packet with the Snort [23] rules in order to filter out the malicious packets. The network packets

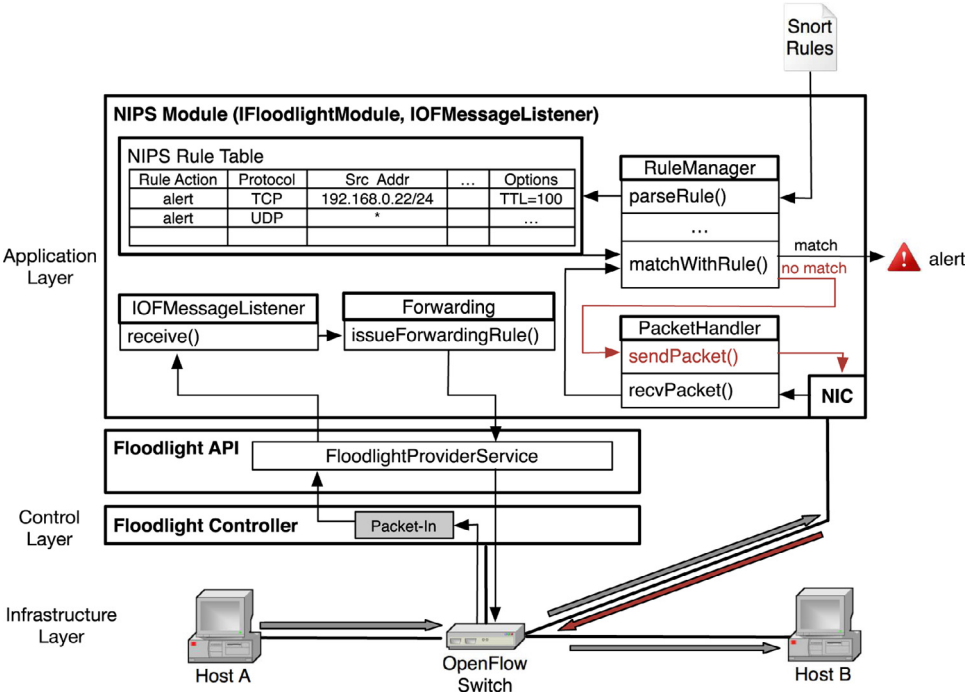


Fig. 4. NIPS application implementation.

that passed the packet inspection process are returned to the network via the network interface. We used the JPcap [24] library to implement the feature of receiving and sending the network packets.

3.3. Discussion

Why in-line mode security functions work in SDN: As has been demonstrated, SDN can turn a network device into a security device that performs access control or intrusion prevention, and it provides three significant benefits. First, additional third-party devices are not required to provide security services. All security functions proposed here are enabled within the network devices, not middleboxes. Second, there is no need to spend time on determining the optimal place for security devices in order to maximize the security service coverage because each SDN-enabled device can function as a security device. Third, we can realize advanced, yet difficult to implement, network security functions with ease. For example, a distributed firewall, which is more effective against internal threats, is typically expensive and complex to implement; however, this function can be implemented on an SDN platform through simply enforcing flow rules to each network device for security purposes.

Why in-line mode security functions do NOT work in SDN: The critical problem might be the SDN performance. Most in-line security functions are required to manage network flows as fast as they can because they should not affect the overall network performance. This issue is carefully investigated in Section 4. Another critical problem might occur when the flow rules for security purposes conflict with the rules for non-security purposes. For example, if an application conducting the network access control function enforces a flow rule to block flows from Host A, and another application that performs network switching function issues a flow rule to forward the same flow at the same time, the data plane will be in an unpredictable state. This issue has been addressed in previous work [15,25,26]; however, we should aware that most systems (i.e. controllers) supporting SDN functions remain vulnerable to such events.

4. Passive mode security applications

We also implemented one passive mode security application in Floodlight, and it was a network intrusion detection system (NIDS).

4.1. NIDS application

The NIDS function is not significantly different from the NIPS function, except that it inspects the network traffic in a passive manner; hence, enabling the NIDS function on an SDN platform confronts similar challenges as those addressed in the NIPS case. However, one difference between the two cases is that the NIDS function does not need to be in-line with the network traffic in order to monitor the traffic. That is, the traffic, which the NIDS wishes to monitor, should be mirrored and forwarded to the original destination and the NIDS concurrently.

Conceptual design: Here, we describe how the NIDS function can be enabled on SDN platforms through presenting a scenario with the same network topology that was used in the scenario from the NIPS case. The NIDS function, which is enabled with SDN, simply compares the network packets with the signature-based rules in order to detect known threats.

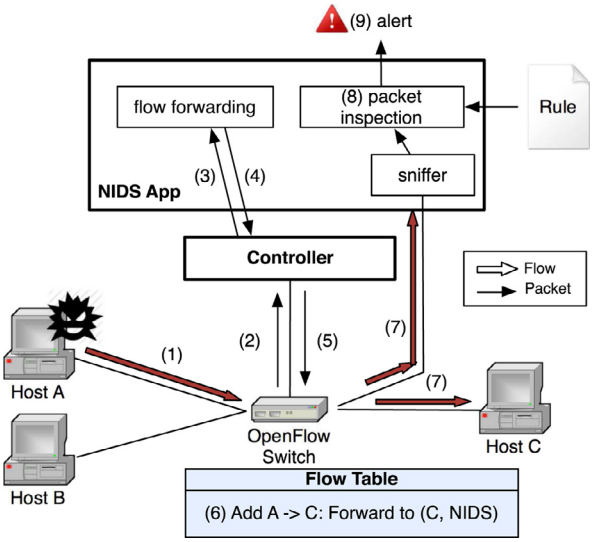


Fig. 5. Conceptual NIDS function implementation.

The NIDS function can be enabled on SDN platforms as depicted in Fig. 5. The NIDS application also initiates as (1) Host A sends a data flow to Host C. Since the flow is unknown to the switch, (2) the first packet of the flow is sent to the controller and (3) it is passed to the NIDS. Then, the NIDS determines if the flow should go through the packet inspection process. When the flow needs to go through the packet inspection, the NIPS notifies the controller that the flow should be forwarded to both its original destination and the NIDS. Next, (5) the controller issues the corresponding flow rule to the switch, and (6) the flow entry of "Add A→C: Forward to C and NIDS" is added to the flow table of the switch. Then, (7) the switch duplicates the flow and concurrently forwards each flow to its destination and the NIDS. Finally, (8) the packets are forwarded to the NIDS for the inspection, and (9) an alert is raised upon detection of malicious traffic.

Floodlight implementation: Unlike the case of the Floodlight NIPS application, the NIDS implementation is straightforward. The overall design of the Floodlight NIDS implementation is almost identical to the design of the NIPS implementation illustrated in Fig. 4. Note that the portions colored in red only apply to the NIPS. For the NIDS, the network traffic should be copied and sent to the NIDS as well as its original destination. In addition to the OpenFlow's capability of the matching ingress port, NIDS also supports the identification of a set of actions to be taken (action set) in a single flow rule. That is, it is possible to forward a matching packet to multiple destinations through sending a single flow rule. Leveraging this OpenFlow feature, the proposed Forwarding module specifies both the original destination and NIDS port number in the flow rule for each *Packet-In* message. Furthermore, unlike the NIPS implementation, the

PacketInspection module for the NIDS only raises an alert for detection.

4.2. Discussion

Why passive-mode security functions work in SDN: Passive mode security applications commonly receive network flows (or its information) from network devices through a mirroring port. This can be undertaken simply through enforcing multiple output actions (i.e. Fig. 5 (6)). Moreover, SDN enables a network device to selectively deliver network flows to a passive mode security application. For example, to monitor web traffic (i.e. flows heading to port 80), the NIDS application can enforce a flow rule that redirects the network flows with destination port 80 to the NIDS, and it can enable the NIDS application to autonomously choose the flow through referring to the network status.

Why passive security functions do NOT work in SDN: As illustrated through our design, passive mode security applications require additional network interfaces between the control plane and the data plane to collect full payload information, which cannot be delivered through a control channel (e.g. an OpenFlow channel). Therefore, this could cause developers to hesitate to apply the SDN technique in implementing their security applications. We believe that SDN needs to provide a method of overcoming this issue, and Avant-Guard [27] may be a good solution. However, to date, we have not found a commercial product that adapts this technique.

5. Network anomaly detection applications

Network anomaly detection functions include detecting network scans and distributed denial of service (DDoS) attacks; these functions are also aggregated in closed and proprietary security appliances in general. We present how these functions can be enabled on SDN platforms and introduce how they were implemented in real Floodlight applications with network scan and DDoS detection functionalities. Unlike the descriptions of other security functions, we only provide a detailed description of the network anomaly detectors design and implementation because the detectors that are presented here only differ in their detection algorithm.

The operations of network scan detectors and DDoS detectors are similar because they both utilize network statistics to detect network anomalies. The ease and flexibility of the anomaly detection algorithm implementation on SDN platforms have been emphasized in previous research [1,14,28]. For example, traditional network anomaly detection appliances (hardware-based) or applications (software-based) manually measure the network statistics through monitoring all network packets; however, through leveraging the SDN, the network statistics can be efficiently measured because this information can be simply retrieved from the data plane.

Conceptual design: The general network anomaly detection based on network statistics can be enabled on SDN platforms as depicted in Fig. 6.

The network anomaly detection application actively monitors the network status through (1) requesting network statistic information from the controller. As requested, (2) the controller sends the request message to reply with the collected network statistic data. Then, (3) the data is sent to the controller, and (4) it arrives at the application. Next, (5) the application analyzes the data in order to extract the information that the detection algorithm requires. Finally, (6) the application makes detection decisions based on the information and (7) raises alerts for the detection.

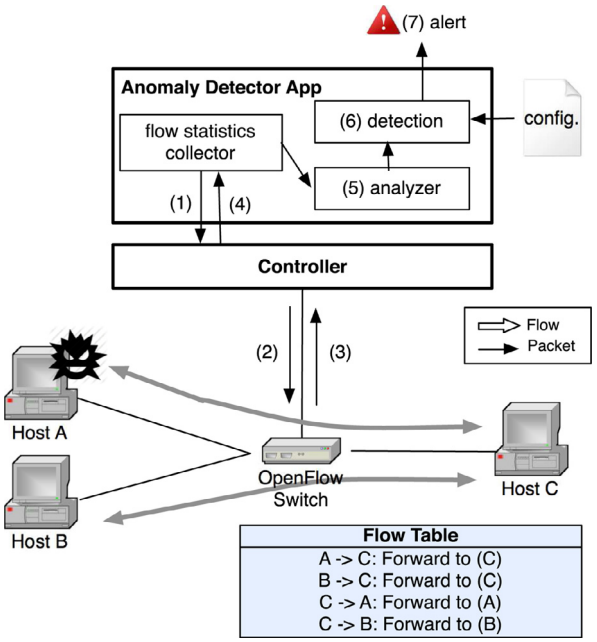


Fig. 6. Conceptual network anomaly detection function implementation.

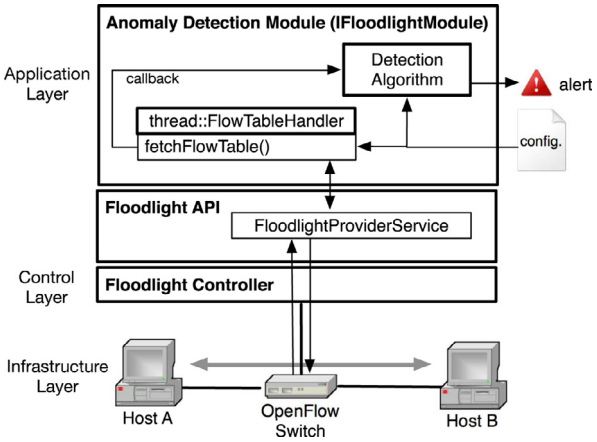


Fig. 7. Network anomaly detection application implementation.

Floodlight implementation: The design of these network anomaly detectors is illustrated in Fig. 7. As mentioned earlier, this design represents both network scan detector and DDoS detector Floodlight applications.

Network anomaly detectors should constantly and periodically collect network statistics, such as byte or packet counts, from the switches on the network. The OF-enabled switches are programmed to keep a record of flow-level network statistics in the flow table. In Floodlight, this data can

be easily fetched via *FloodlightProviderService*. In our implementation, we created a thread that periodically requested the flow table data from the controller within the application (*FlowTableHandler* in Fig. 7). The flow table collection period can also be supplied via the configuration file. Shortly after the data collection process, the application aggregates and analyzes the collected data in order to detect anomalies. The constant values for the detection algorithm can also be supplied through the configuration file.

5.1. Scan detector

We implemented the anomaly score [29] based scan detection algorithm in our application. Upon the collection of the network statistic information from the switch, the application analyzed the collected data in order to compute the anomaly score for each destination port. The pseudocode of the algorithm is provided below.

```
function scan_detection(stats){
    foreach(stat : stats){
        port_map[ stat.getDstPort() ] += 1
        counts += 1
    }
    foreach(port : port_map){
        prob = port_map[port] / counts
        learned_prob = learned_port_map[port]
        anomalyscore += -log_2[prob/learned_prob]
    }
    if(threshold < anomalyscore){
        alert();
    }
}
```

5.2. DDoS detector

The algorithm implemented in the DDoS detection application simply maintains a count of the number of bytes and packets in order to derive the byte rate and packet rate from the collected data. Its pseudocode is presented below in order to provide a better understanding of the DDoS detection mechanism that was used.

```
function ddos_detection(stats){
    foreach(stat : stats){
        bytes += stat.getByteCount()
        counts += stat.getPacketCount()
    }
    bytes -= previous_bytes;
    counts -= previous_counts;

    bps = bytes/time_interval
    pps = counts/time_interval

    previous_bytes = bytes;
    previous_counts = counts;
    if(threshold_bps < bps|threshold_pps < pps){
        alert();
    }
}
```

5.3. Discussion

Why network anomaly detection functions work in SDN: We believe that this type of application is representative of network security functions that benefit the most from SDN. Network anomaly detection systems commonly require a device (or a method) to collect network status information, and this often places a burden on the network administrator. However, if SDN functions are enabled in the network, the network administrator does not need to collect the information, because the detector can easily capture the network status (even fine-grained flow information) without additional devices or complicated configurations.

Why network anomaly detection functions do NOT work in SDN: Indeed, through the SDN function, we can collect network status information easily. However, we sometimes may need more information that is difficult to collect in the current SDN environment; TCP session information is a good example of this. We cannot collect this information through simply sending a query to the data plane, but rather it requires the enforcement of a set of flow rules (at least more than two consecutive flow rules) on the data plane in order to obtain this information. Furthermore, we must consider how much overhead will be added to the data plane through periodically sending this query to the data plane. These issues will be discussed in Section 7.

6. Other advanced network security applications

For advanced security functions, we implemented the stateful firewall and reflector network functions in the Floodlight application. In this section, we describe the design and operation of the stateful firewall and reflector network function implementations in detail.

6.1. Stateful firewall

Leveraging the programmability of SDN platforms, it is possible to implement a more advanced firewall function. In addition to the bundle firewall (stateless), we also introduce the design and implementation of a stateful firewall function that operates on SDN platforms.

Conceptual design: ACL-based stateless firewalls (discussed in Section 3.1) offer somewhat limited network usability and high configuration complexity. In order to overcome such disadvantages, stateful firewalls that dynamically track the state of each connection have been proposed. For example, a stateless firewall is incapable of supporting file transfer protocol (FTP) operations because these protocols open TCP connections to arbitrary high ports. In contrast, a stateful firewall dynamically tracks the state of the valid connections and allows any derived connection; thus, it is capable of supporting the FTP. Fig. 8 conceptually illustrates

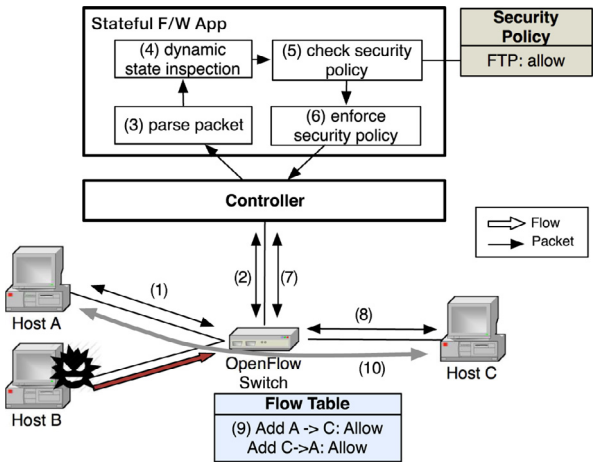


Fig. 8. Conceptual stateful firewall design.

how the stateful firewall function can be enabled in SDN platforms.

In order to elaborate on the operation of the stateful firewall in SDN platforms, we demonstrate how the firewall allows an FTP connection in the network depicted in Fig. 8. We assume that the firewall enforces the security policy that only allows FTP connections to the network. When (1) Host A attempts to make an FTP connection (or sends an FTP request packet) to Host C for the first time, (2) the OpenFlow-enabled switch passes the packet to the controller. Once the firewall application captures the packet, it (3) parses the packet and (4) compares it with the packet in the state table, which tracks every valid connection, in order to verify if the packet belongs an existing connection being tracked in the table. If the packet is a request packet that attempts to open a new FTP connection, it is added to the state table as a new table entry with the status indicating that it is awaiting a response from the FTP server. If the packet neither belongs to an existing connection nor a new connection request packet, it is then (5) examined to determine if it is against the predefined security policy. If the packet violates the policy, no action is taken (or the packet is dropped). In contrast, if the packet does not violate the policy, (6) the firewall application installs appropriate flow rules to allow the connection. When the packet (FTP request packet) has passed the inspection, (7, 8) it is forwarded to its destination (Host C). In response to the request, (8) Host C replies with an arbitrary port number for the data transfer. Since there is no matching rule in the flow table of the switch, the response packet is sent to the controller, and it undergoes the same firewall operation (3, 6). During these steps, the firewall learns that the response packet belongs to the existing connection, and it also learns the port number that will be used for the FTP data transfer. Accordingly, (9) the firewall can allow the FTP connections through installing two flow rules with the dynamically allocated port number specified to the flow table. Ultimately, unlike the stateless firewall, the stateful firewall function can (10) allow complete and flawless FTP operations with SDN.

Floodlight implementation: As stateful firewalls track the state of each connection, they must operate at the top layer of the seven-layer OSI model. Meanwhile, as mentioned above, due to the limited capability of OpenFlow, the higher layers (i.e. higher than layer 4) are invisible to the control plane. Hence, in order to enable stateful inspection, the stateful firewall implementation uses the design that utilizes an additional network interface (as introduced in the NIPS/NIDS implementations) as a side channel. Fig. 9 illustrates the design of our Floodlight stateful firewall implementation.

Our stateful firewall application also operates in a reactive manner, which is the same as the stateless firewall application. Once the application receives a *Packet-In* message, the *PacketParser* module parses the message to extract the necessary information, such as IP protocol, IP addresses, sequence numbers, and TCP port numbers involved, and it passes the information to the StateManager. In our implementation, the stateful inspection is only performed on TCP connections. Accordingly, for each *Packet-In*, the *StateManager* determines if the incoming packet is TCP-based or not. For non-TCP packets, the *StateManager* simply tests if the packet matches the ACL and installs the appropriate flow rules (via *FLOW_MOD*) in order to enforce the security policy. Meanwhile, for TCP packets, the *StateManager* notifies the *Forwarding* module to forward the corresponding raw packet to the side channel for state inspection. Unlike how the firewall forwards non-TCP packets, each TCP packet is forwarded using *Packet-Out* messages in order to successfully track the states of the connection through monitoring every packet in the connection. Each connection information and its state are stored on *StateTable*, which is a simple data table maintained by the *StateManager*. The *StateManager* analyzes each raw packet that arrives via the side channel and updates the corresponding *StateTable* entry for the existing connections. For a new connection, the *StateManager* registers a new StateTable entry with the state of the connection completed. For example, an SYN packet would be analyzed and registered to the *StateTable* as a new table entry with its state flagged. The state flag indicates that the firewall is expecting a legitimate SYN-ACK packet. In some cases, like the FTP protocol, protocol-specific states are defined (e.g. directory listing requests, awaiting the directory listing, etc.). When an FTP client connects to an FTP server, the server responds with an arbitrary port number that is then used to transfer data. Our firewall application is designed to parse these protocol-specific responses and dynamically allow connections.

6.2. Discussion

Why stateful firewall function works in SDN: Stateful firewall functions are generally implemented in closed and proprietary appliances. For real deployment, there are many issues to be considered, including the cost, space, operational cost, and so on. With SDN, it is possible to implement the same functionality in an SDN application as described above. The stateful firewall function can be implemented at a low cost, and it does not require additional space for deployment. The centralized structure also reduces the operational cost. Furthermore, the stateful firewall function may require frequent firmware updates to support various protocols (e.g. FTP). In this form of closed and distributed device, it is often difficult to update, reconfigure, and maintain the appliances. In contrast, SDN applications provide simple and convenient environments to perform such tasks.

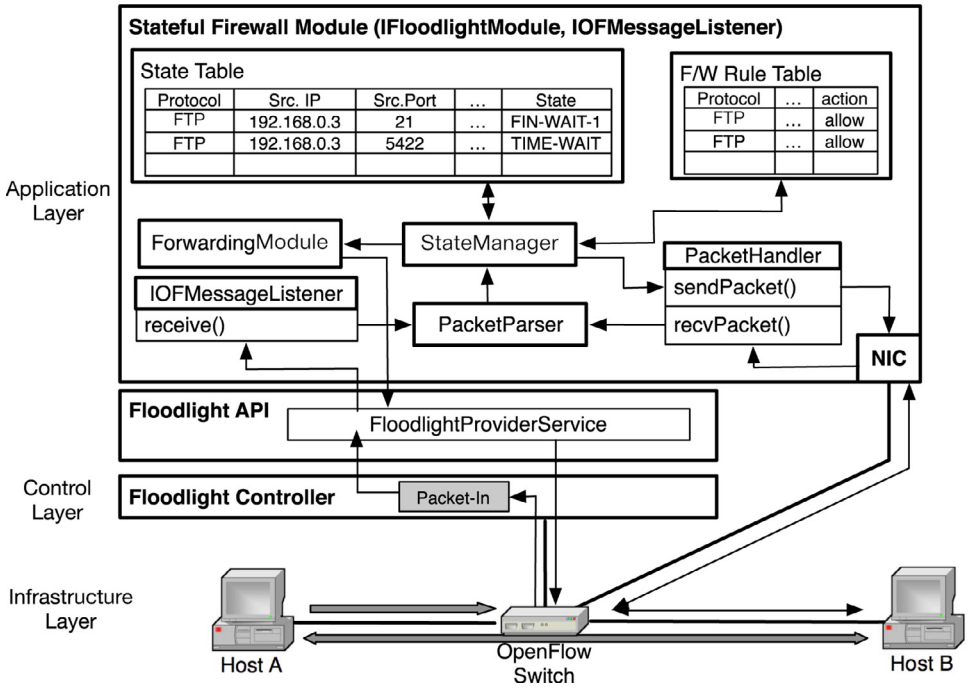


Fig. 9. Stateful firewall application implementation.

Furthermore, using the stateless firewall, it is possible to realize a distributed stateful firewall that is effective against internal as well as external threats.

Why stateful firewall function does NOT work in SDN: When enabling stateful firewall functions with SDN, one of the most important issues that need to be addressed is the impact on network performance. As the stateful firewall function operates in-line with the traffic, it directly affects the network performance. As described in the implementation, it uses an additional network interface to retrieve the raw packet that is involved in the connection that is being monitored. Although such design incurs additional control path delays as each packet travels to the control plane twice, it is inevitable due to the limited capability of OpenFlow. The network performance impact of our stateful firewall application is discussed further in Section 7.3.

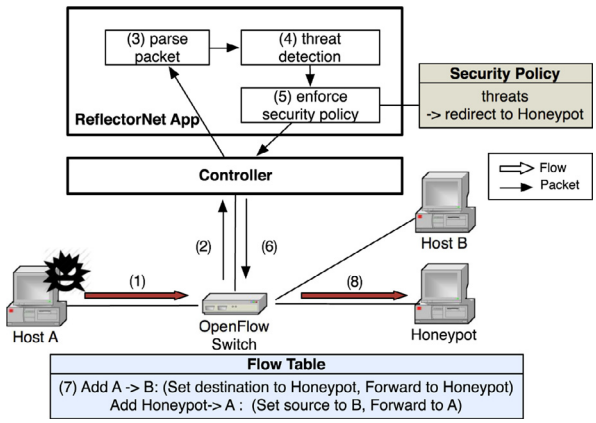


Fig. 10. Conceptual ReflectorNet function implementation.

6.3. Reflector network

A reflector network (ReflectorNet) function redirects any threats to a honeypot upon detection. This function can be deployed to collect forensic evidence for various purposes. Realizing the ReflectorNet function in legacy network platforms is difficult, because the function requires a substantially flexible and programmable network infrastructure to redirect certain traffic successfully. On an SDN platform, this function can be effectively enabled through leveraging the OpenFlow capabilities of modifying and forwarding packets.

Conceptual design: We present an example scenario that describes how the ReflectorNet function can be enabled on an SDN platform. The scenario involves three hosts (A, B, and honeypot) connected to an OpenFlow-enabled switch as illustrated in Fig. 10.

As depicted in Fig. 10, the scenario begins with (1) Host A sending data traffic to Host B, and (2) the switch transfers the first packet of the flow to the controller. Then, (3) the ReflectorNet application parses the packet to extract any information that the detection algorithm requires. Next, (5) the ReflectorNet requests the controller to redirect the flow to the honeypot. Then, (6) the controller issues the corresponding flow rules, and consequently (7) the flow entries that are redirected and the reverse flow are added to the flow table. Finally, (8) the data traffic arrives at the honeypot as though the honeypot was Host B.

Floodlight implementation: The design of our Floodlight ReflectorNet application is presented in Fig. 11.

Similar to the other security applications introduced previously, the Floodlight ReflectorNet application also includes

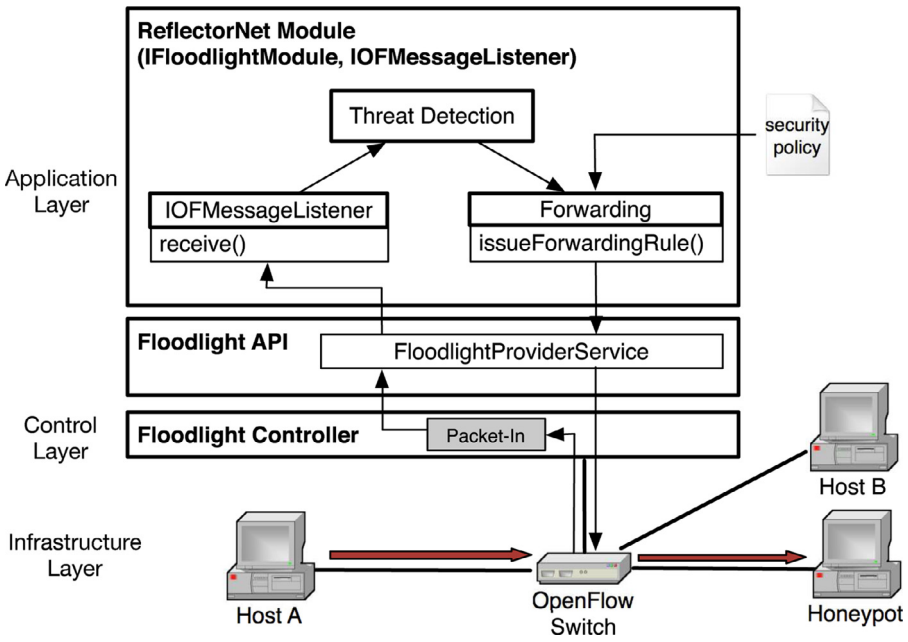


Fig. 11. ReflectorNet application implementation.

the *IOFMessageListener* to capture the *Packet-In* messages. The ReflectorNet initiates upon the arrival of a *Packet-In* message and the *ThreatDetection* module analyzes the message in order to verify if the flow is a threat. Instead of employing the scan detection module that already exists, the *ThreatDetection* module was implemented to behave as a simple blacklist-based detector to accurately evaluate the key feature of the ReflectorNet. The key feature is implemented in the *Forwarding* module. The Forwarding module aims to manipulate both the source-to-destination and destination-to-source flows in order to redirect each path of the connection, and this manipulation task can be effectively accomplished using OpenFlows set action. For the source-to-destination flow, the set action is used to replace the destination field of the packet header with the location of the honeypot, and the output action is used to forward the flow to the honeypot. The Forwarding module specifies both actions as an action set in a flow rule that is issued. The reverse connection is managed in a similar way; thus, in order to enable the ReflectorNet function, the module only installs two flow rules for each detected *Packet-In* message.

6.4. Discussion

Why ReflectorNet function works in SDN: Through writing only 400 lines of code, we were able to implement a ReflectorNet function. This implies that such advanced network security function, which may require an additional hardware device to enable in the traditional network, can be easily realized by taking advantage of the capability of SDN. Currently, SDN provides numerous interesting techniques to manage network flows, and it enables an application to have a network-wide view. These features are important when a network security function is designed, because they can reduce the implementation burden. For example, if the network devices in an enterprise network should be monitored, then a sensor should be installed on each network device, which is a tedious task. However, if we consider SDN, the network application can see all network statuses through simply sending a query to each network device. Likewise, the features that SDN provides enable the creation of complicated security functions with less effort.

Why ReflectorNet function does NOT work in SDN: Although SDN provides numerous interesting features, it cannot be the only solution. Some interesting features, such as the QoS control of network flows, have not yet been implemented in real devices, and thus we should carefully consider whether the selected SDN features for implementing security functions are currently available. In addition, the current SDN functions have not critically considered the supporting security functions. Therefore, although they can be easily added to the current SDN architecture, many required features are currently missing, and thus we (i.e. security-related people) need to provide feedback to the wider SDN community.

7. Evaluation

In this section, we evaluate the Floodlight security applications in order to provide clear answers to the questions raised in Section 2. We measure the performance of each application in order to prove the feasibility of security applications on an SDN platform.

7.1. Experimental setup

In order to evaluate our work, we have constructed three physical SDN testbeds, each consisting of an OpenFlow-enabled switch, a controller machine, and three physical hosts. Three OpenFlow-enabled switches (HP 3500yl, 3800

Table 1
Comparison of OF-switch performances.

	HP 3500yl [30]	HP 3800 [31]	Pica8 P-3290 [32]
Switch fabric capacity	101.8 Gbps	88 Gbps	176 Gbps
Forwarding speed	75.7 Mpps	65.4 Mpps	132 Mpps
Latency	3.4 us	2.8 us	1 us
Routing table size	10,000	10,000	12,000
MAC table size	64,000	65,500	32,000

Table 2
Specifications of the machines deployed in the testbeds.

Type	NIC	CPU	RAM	OS
Controller	1 Gbps × 5	i5-4570	16 Gb	Ubuntu 12.04 64 bit
Host 1 (H1)	1 Gbps	i7-2640 M	8 Gb	Ubuntu 12.04 64 bit
Host 2 (H2)	1 Gbps	i5-2450 M	8 Gb	Windows 7 64 bit
Host 3 (H3)	100 Mbps	Atom N550	2 Gb	Ubuntu 13.10 64 bit

and Pica8 P3290) were deployed in each testbed, and the specifications of each switch are provided in Table 1.

We evaluate our work on three distinct testbeds with different switches in order to avoid biased performance measurements that might result from switch-specific factors. The specifications of the controller and host machines deployed in the testbeds are described in Table 2.

In order to demonstrate the feasibility of various security applications on SDN platforms, we measured the throughput of each application for different data rates on each testbed. For the experiment, we semi-randomly generated data traffic for the applications that involve packet-matching processes (firewall, NIDS, and NIPS). More specifically, we intentionally controlled the data traffic in order to not include a specific port number. Furthermore, we also generated the firewall and Snort rules to include this port number so that the application would be forced to match a packet against every rule that exists in the given ruleset. For the remainder of the applications, we randomly generated data traffic to measure the throughput. The traffic was sent from H1 to H2, and we measured the number of packets received at H2. Unlike other applications, the evaluation of the ReflectorNet application involved the three hosts. In this case, the traffic was sent from H1 to H2; however, the number of delivered packets were measured at H3.

For the stateful firewall application, we measured the latency incurred during a specific FTP operation for one testbed (HP 3500yl) and compared this result with the latency incurred during a simple learning switch application as a baseline. For the latency experiment, we configured H1 to attempt an FTP login and directory listing request to H2 (FTP server), and we measured the time elapsed to complete the FTP operations for each case.

7.2. Throughput evaluation

For each Floodlight security application, we plotted our measurements in order to illustrate how the throughput of the switches varies with respect to the data rate.

7.2.1. In-line mode security application

Firewall application: Fig. 12 demonstrates that the throughput of the switch substantially degrades as the data rate increases when the firewall is deployed. The firewall included in the Floodlight distribution is designed to inspect every network packet. For this reason, a throughput is observed for the higher data rates. Instead of sending a *Flow_Mod* message that inserts a flow rule into the flow table, the firewall sends a *Packet-Out* message to forward each packet. The purpose of this design can be inferred from the firewalls user interface. As depicted in Fig. 2, the firewall policy can be dynamically changed via the REST API. We presume that the *Packet-Out* message is used to effectively enforce the dynamically changing firewall policy to every packet. Overall, Pica8 outperformed both HP devices in this case. Fig. 12 also shows how the number of firewall rules loaded on the firewall (1000, 5000, and 10000 rules) affects the throughput, and it can be implied that the number of firewall rules does not noticeably affect the performance of the switch.

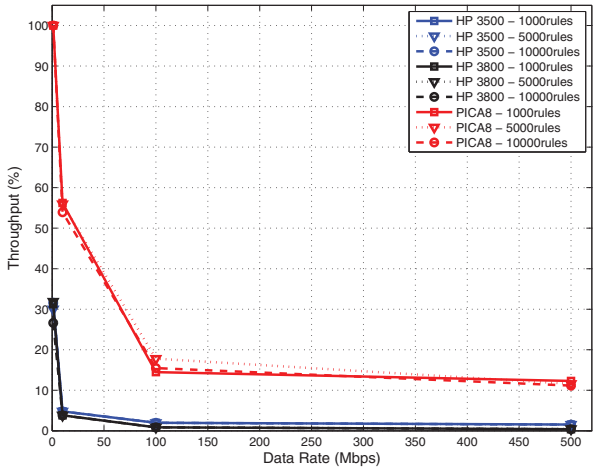


Fig. 12. Firewall application: throughputs for varying data rates and the number of firewall rules.

NIPS application: The result of the NIPS application test in Fig. 13 indicates that the NIPS application is capable of managing the traffic with a data rate up to 100 Mbps with 1000 Snort rules. Compared with the firewall that does not

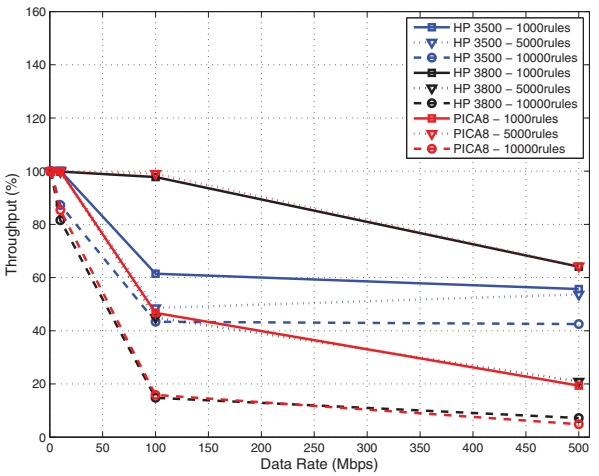


Fig. 13. NIPS application: throughputs for varying data rates and the number of Snort rules.

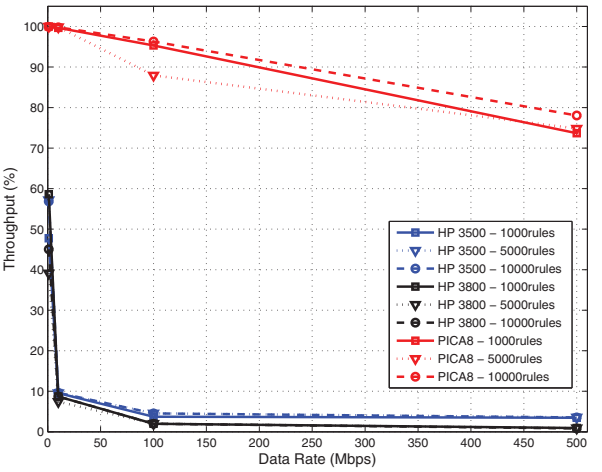


Fig. 14. NIDS application: throughputs for varying data rates and the number of Snort rules.

install flow rules, the NIPS outperforms it because the NIPS installs flow rules for each incoming flow and thereby receives significantly fewer *Packet-In* messages to handle. However, a problem arises when the application needs to match packets against more than 5000 Snort rules.

Discussion: We expected that the payload delivery from the data plane to the control plane would incur substantial overhead; however, throughout the experiment, it was found that it does not add significant overhead, but rather the *Packet-In* messages affect the system. This result implies that if our network does not produce new flows, causing *Packet-In* messages, and there are less than 1000 matching rules, we can use the SDN technique in deploying an in-line style security function. Hence, the networks that mostly carry long-lived network flows, or that only requires the protection of the network flows for certain critical services (e.g., web, mail, or other services) may benefit from deploying SDN-based in-line mode security applications. In addition, Buchanan et al. demonstrated that the throughput of the in-line mode Snort on a non-SDN platform substantially degraded when more than 5000 Snort rules were loaded [33].

7.2.2. Passive mode security application

NIDS application: Just like the firewall evaluation, we have measured the throughput of each switch device at different data rate and different number of Snort rules. In the NIDS application, a significant performance gap between the two switches was observed as depicted in Fig. 14. While Pica8 achieved a throughput of at least 70% up to a data rate of 500 Mbps, HP devices barely achieved a throughput of less than 5%, regardless of the number of NIDS rules loaded by the application. In order to better understand the issue, we examined how each device inserted the flow rules into their tables. We found that Pica8 maintained the flow rules in its hardware table, while HP loaded the software flow table when it needed to manage multi-forwarding rules (i.e. send a packet to multiple output ports). Although only low rate of network traffic was injected in our experiment, this software-based flow table did not perform very well, and thus we recommend that this type of device is not used in this situation.

Discussion: Some devices try to manage SDN specialized functions in software, and this will cause too much overhead. However, if we can guarantee that SDN-enabled devices handle all (or at least critical) functions in the hardware, then we can use them to implement practical network security functions. Alserhani et al. demonstrated that Snort achieved a throughput of 70% for data rates from 500 Mbps to 700 Mbps on a non-SDN platform [34]. Therefore, we can infer that our NIDS application with the Pica8 switch is able to perform as well as Snort on a non-SDN platform.

7.2.3. Network anomaly detection application

In order to evaluate our network anomaly detection application, we measured a throughput of each switch device for different data rate and different flow statistic query interval.

Scan detection application: The performance of the scan detection application did not appear to be affected by the flow table collection interval, rather the measurements indicated that it depended on the performance of the switch as depicted in Fig. 15. This clearly demonstrates that using SDN in network status monitoring is a reasonable idea, and practical network anomaly detectors can be designed if they rely on the information provided by the SDN functions.

DDoS detection application: As seen in Fig. 16, the DDoS test results also depict similar results to those from the scan detection. This is natural because the detector uses similar features as the scan detector, and the only difference is the decision mechanism whose overhead is minimal.

Discussion: SDN functions provide different network status information, such as received packet counts of a flow, and our experiments demonstrate that collecting this information does not add significant overhead. If a network anomaly detector (or other security sensor) is based solely on this collection, the detector can be used in real-world environments.

7.2.4. Advanced security application

Reflector network application: For the reflector network application, the throughput of the switches substantially

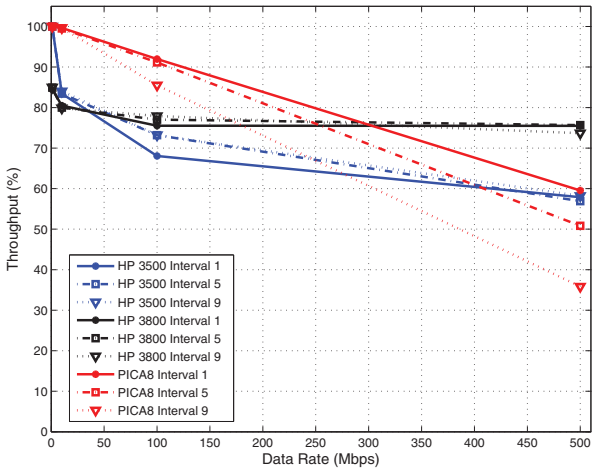


Fig. 15. Scan detection application: throughputs for varying data rates and flow statistics query interval (in seconds).

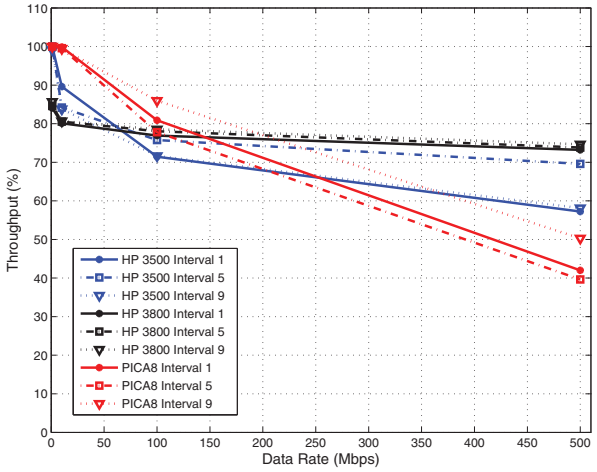


Fig. 16. DDoS detection application: throughputs for varying data rates and flow statistics query interval (in seconds).

degraded as the data rate increased. Although Pica8 outperformed HP devices, both cases did not exhibit sufficient performance that could be applied in real environments. For the data rate of 1 Mbps, Pica8 achieved 100% throughput while HP devices achieved 30%, as seen in Fig. 17. However, for the data rate of 100 Mbps, both switches achieved a throughput of less than 20%. This degradation also resulted from the handling flow rules in the software. We found that all the devices inserted SET actions, which modified the packet headers, into their software rule table instead of the hardware table.

Discussion: Modifying the packet headers is one of the most important features of SDN, and many previous studies have proposed interesting and advanced network or security functions [7,8,35]. However, if most network devices only implement this feature in the software layer, the proposed ideas cannot be realized in real cases because the performance becomes another critical problem in many cases.

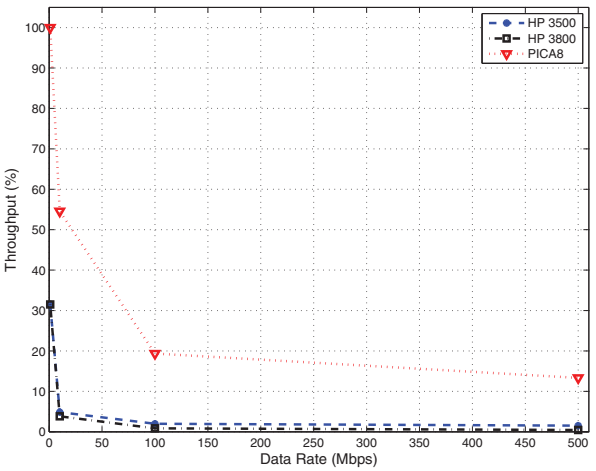


Fig. 17. ReflectorNet application: throughputs for varying data rates.

Table 3
Latency evaluation.

	Learning switch	Stateful firewall
Avg. latency (ms)	20.676	31.595

7.3. Latency evaluation

Stateful firewall application: For the stateful firewall application, the latency incurred during a legitimate FTP connection establishment was measured. We also measured the latency impact of the learning switch application, which comes with the Floodlight distribution, as a baseline to evaluate our work.

As seen in Table 3, compared with the baseline learning switch application, the performance overhead (in terms of latency) that our stateful firewall application added was only approximately 11 ms. Considering that the stateful firewall performs packet-by-packet inspections and dynamic state tracking at the control plane, we believe this overhead could be overcome through adding more computing power to the controller machine.

7.4. Insights

Based on the findings from our experiments, we provide some insights that should be noted.

1. Reducing the number of control messages from the data plane to the control plane (i.e., *Packet-In*) is much more important than minimizing the delivery of packet payload to the control plane. This event-driven control mechanism is the basic principle of SDN, and thus it is difficult to reduce the number of control messages. Therefore, to implement and deploy time-critical security functions on SDN platforms, we should consider another solution such as a distributed control platform (e.g., Onix [36], ONOS [37]) and a high-speed control platform (e.g., Beacon [38]).
2. Each data plane can provide interesting network status information (e.g., bytes sent/received), and this implies

that the approach of regarding data planes as a type of database is a feasible idea [39]. In the recent OpenFlow specification, the data plane maintains diverse network status information, and the information is available to the SDN applications on the control plane. Leveraging such capability of OpenFlow, it is possible to implement the SDN applications that investigate network status.

3. Examining the supported SDN functions of each data plane is very important. In particular, if performance is important, the required functions that are supported at the hardware level (i.e. provide reasonable performance) should be considered carefully. In our experiments, we showed that each network device does not necessarily achieve the expected throughputs according to their general performance specifications (i.e., forwarding speed, latency, etc.) when the security applications were deployed. Those who consider implementing or deploying SDN-based security applications may find our work significantly useful. For example, our work shows that the switch devices comprising the network should support hardware-based multi-forwarding to sufficiently secure the network with SDN-based NIDS application.
4. There have been a few attempts to improve (or modify) the existing SDN functionalities for significantly improving the performance of security functions or increasing the ease of development of the functions. For example, Avant-Guard [27] extends the data plane to increase the performance, scalability and resiliency of SDN-based security services, and VeriFlow [40] adds an extra layer between the control and data planes to detect network-wide invariant violations. To date, SDN techniques are mostly leaded by network communities, and thus security of SDN is often not considered. We hope security researchers actively dive into SDN and flourish the security functionalities of SDN.
5. It is possible to implement in-line mode security functions in SDN applications, and they are effective in some particular networks that aim to provide security services for a small number of network flows. Passive mode security applications can be also useful if the network devices comprising the managed network handle the required SDN functions in the hardware. (In our experiment, the Pica8 device supported hardware matching for multi-port-forwarding, and outperformed the HP devices.) In the case of network anomaly detection applications, fetching network status information via SDN function do not incur significant overhead, and therefore implementing such type of security functions in SDN applications is a feasible idea. Meanwhile, the advanced security applications (Reflector network) that attempt to leverage SDN's capability of modifying packet header information are only effective if the network devices process such modification in the hardware.
6. We also (partially) elucidate the feasibility of enabling various security functions with Network Functions Virtualization (NFV) technology. NFV is also an exciting technique that enables the delivery of useful network services via virtualization technology, and it has a lot in common with SDN. Particularly, in a sense that our security applications provide useful network security functions with no hardware dependencies, they can also be considered as NFV applications. Thus, those who consider providing security services in an NFV environment may benefit from our experience as well.

8. Related work

Although most current SDN research is highly focused on network related topics (e.g. network management and network routing), there is some pioneering research relating to security issues with SDN. FortNOX presents the security problems of SDN (e.g. dynamic flow tunneling) and their solution [25]; Shin et al. revealed that a type of network flooding attack is feasible in an SDN network [41]; and Kreutz et al. summarize the possible security problems in SDN [42]. These studies have provided valuable information that assist SDN to become more secure. However, our goal differs significantly: the main goal of this work is to investigate the possibility of designing practical security functions with SDN. Huang et al. benchmarked three OpenFlow-enabled switch models from different vendors and compared them in order to demonstrate how the switch implementation design impacts network performance [43]. Again, our work differs to theirs because our experiment involves not only OpenFlow-enabled switches but also security applications. We focus on how different security applications affect the overall throughput/latency with different switches in order to clearly demonstrate the advantages and disadvantages of enabling each security functions with SDN.

SDN techniques have also been applied in realizing network security functions. In order to detect network flooding attacks (e.g. DDoS attacks), Braga et al. proposed a lightweight detector and they implemented the proposed idea of the NOX platform [1]. Furthermore, OpenSketch provides an efficient network flow measurement scheme [22]. Mehdi et al. evaluated several algorithms in order to understand if they could effectively detect network scan attacks [14]. FleXam designed a flexible sampling extension for OpenFlow and demonstrated that it could detect a port scan attack with an extremely low overhead [28]. Moreover, SDN functions can be used in building different types of security functions. Hu et al. presented a possible solution to make a reliable firewall using the SDN technique [44], and vArmour announced that they will release a dynamic firewall based on SDN functions [16]. Moreover, a new framework for developing security applications using SDN has been recently proposed [8]. In addition, Avant-Guard improves the current SDN architecture to enable SDN to provide better security services [27]. These studies illustrate that SDN can be used for security purposes. However, they only present conceptual ideas or research prototypes; thus, it is very difficult to understand whether SDN really helps in developing practical security functions. Our work begins investigating this issue, and we attempted to discover the effectiveness, feasibility, and efficiency of security functions based on SDN techniques.

9. Conclusion and future work

While the networking community has focused on SDN and considers it as a promising future networking technology, the security community is relatively slow in embracing SDN technology. Many studies have adapted SDN to resolve

networking issues, and it remains as ongoing work. However, considering SDN in developing network security functions remains in the early stages, and it is difficult to find case studies that examine the feasibility, practicability, effectiveness, and efficiency of network security applications based on SDN technology. Our work does not introduce new security functions; however, we present how the current security functions could be changed in a new networking era (i.e. the SDN network environment) through our experience of implementing several network security functions with SDN. We believe that the findings and insights discussed here can encourage security researchers to devise more and better security functions. In our future work, we will implement more network security applications in SDN and investigate if there are further issues that should be considered. In addition, we will deploy our applications in the real world, which will provide greater insights.

Acknowledgment

This work was supported by the ICT R&D program of MSIP/IITP (grant no. 2014 044-072-003 Development of Cyber Quarantine System using SDN Techniques).

References

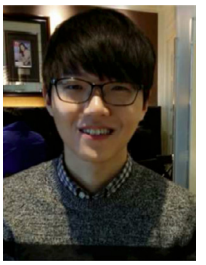
[1] R.S. Braga, E. Mota, A. Passito, Lightweight DDoS flooding attack detection using NOX/OpenFlow, in: Proceedings of the 35th Annual IEEE Conference on Local Computer Networks, in: LCN, 2010.
[2] M. Canini, D. Venzano, P. Perešíni, D. Kostić, J. Rexford, A NICE way to test OpenFlow applications, in: USENIX Symposium on Networked Systems Design and Implementation, 2012.
[3] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, N. McKeown, Elastictree: Saving energy in data center networks, in: Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI, 2010.
[4] A. Nayak, A. Reimers, N. Feamster, R. Clark, Resonance: dynamic access control for enterprise networks, in: Proceedings of WREN, 2009.
[5] L. Popa, M. Yu, S.Y. Ko, I. Stoica, S. Ratnasamy, CloudPolice: taking access control out of the network, in: Proceedings of the 9th ACM Workshop on Hot Topics in Networks, HotNets, 2010.
[6] R. Sherwood, G. Gibb, K.K. Yap, G. Appenzeller, Can the production network be the testbed, in: Proceedings of USENIX Operating System Design and Implementation, OSDI, 2010.
[7] S. Shin, G. Gu, CloudWatcher: Network security monitoring using OpenFlow in dynamic cloud networks (or: How to provide security monitoring as a service in clouds?), in: Proceedings of the 7th Workshop on Secure Network Protocols (NPSec'12), co-located with IEEE ICNP'12, 2012.
[8] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, M. Tyson, FRESCO: Modular composable security services for software-defined networks, in: Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13), 2013.
[9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, OpenFlow: Enabling innovation in campus networks, SIGCOMM Comput. Commun. Rev. 38 (2008) 69–74.
[10] ONF: Open Networking Foundation, https://www.opennetworking.org/.
[11] NetworkWorld, Gartner: 10 critical IT trends for the next five years, http://www.networkworld.com/news/2012/102212-gartner-trends-263594.html.
[12] M.T. Review, 10 emerging technologies: Tr10: software-defined networking, http://www2.technologyreview.com/article/412194/tr10-software-defined-networking/.
[13] EnterpriseNetworking, IDC: SDN a $2 billion market by 2016, http://www.enterprisenetworkingplanet.com/datacenter/idc-sdn-a-2-billion-market-by-2016.html.
[14] S. Mehdi, J. Khalid, S. Khayam, Revisiting traffic anomaly detection using software defined networking, in: Recent Advances in Intrusion Detection, 2011.
[15] H. Hu, W. Han, G.-J. Ahn, Z. Zhao, FlowGuard: Building robust firewalls for software-defined networks, in: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, HotSDN '14, ACM, New York, NY, USA, 2014, pp. 97–102, doi:10.1145/2620728.2620749.
[16] vArmour, http://www.varmour.com/.
[17] FloodLight, Open SDN controller, http://floodlight.openflowhub.org/.
[18] Big Switch Networks, http://www.bigswitch.com/.
[19] R. Wang, D. Butnariu, J. Rexford, OpenFlow-based server load balancing gone wild, in: Proceedings of Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services, HotICE, 2011.
[20] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, R. Wattenhofer, Achieving high utilization with software-driven wan, in: Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, SIGCOMM '13, ACM, New York, NY, USA, 2013, pp. 15–26, doi:10.1145/2486001.2486012.
[21] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, A. Vahdat, B4: Experience with a globally-deployed software defined WAN, in: Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, 2013.
[22] M. Yu, L. Jose, R. Miao, Software defined traffic measurement with OpenSketch,
[23] Snort, Open source network intrusion detection system, http://www.snort.org/.
[24] JPcap, A network packet capture library for Java, http://jpcap.sourceforge.net/.
[25] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, G. Gu, A security enforcement kernel for OpenFlow networks, in: Proceedings of the First Workshop on Hot Topics in Software Defined Networks, HotSDN '12, 2012.
[26] Z.A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, M. Yu, SIMPLE-fying middlebox policy enforcement using SDN, in: Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, ACM, New York, NY, USA, 2013, pp. 27–38, doi:10.1145/2486001.2486022.
[27] S. Shin, V. Yegneswaran, P. Porras, G. Gu, 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'13), 2013.
[28] S. Shirali-Shahreza, Y. Ganjali, Efficient implementation of security applications in OpenFlow controller with flexam, in: IEEE 21st Annual Symposium on High-Performance Interconnects, 2013.
[29] C. Krügel, T. Toth, E. Kirda, Service specific anomaly detection for network intrusion detection, in: Proceedings of the 2002 ACM Symposium on Applied Computing, SAC '02, ACM, New York, NY, USA, 2002, pp. 201–208, doi:10.1145/508791.508835.
[30] HP, HP 3500 and 3500yl switch, http://h17007.www1.hp.com/us/en/networking/products/switches/HP_3500_and_3500_yl_Switch_Series/index.aspx.
[31] HP, HP 3800 switch, http://h17007.www1.hp.com/us/en/networking/products/switches/HP_3800_Switch_Series/index.aspx.
[32] Pica8, Data sheet: Pica8 P-3290, http://www.pica8.com/documents/pica8-datasheet-48x1gbe-p3290-p3295.pdf.
[33] W. Buchanan, F. Flandrin, R. Macfarlane, J. Graves, A methodology to evaluate rate-based intrusion prevention system against distributed denial-of-service (DDoS), in: Cyberforensics, 2011.
[34] F. Alserhani, M. Akhlaq, I.U. Awan, A.J. Cullen, J. Mellor, P. Mirchandani, Snort performance evaluation, in: Proceedings of Twenty-Fifth UK Performance Engineering Workshop, 2009.
[35] J.H. Jafarian, E. Al-Shaer, Q. Duan, OpenFlow random host mutation: transparent moving target defense using software defined networking, in: Proceedings of the First Workshop on Hot Topics in Software Defined Networks, HotSDN '12, 2012.
[36] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, S. Shenker, Onix: a distributed control platform for large-scale production networks, in: The Symposium on Operating Systems Design and Implementation (OSDI), 2010.
[37] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, et al., ONOS: towards an open, distributed SDN OS, in: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, ACM, 2014, pp. 1–6.
[38] OpenFlowHub, BEACON, http://www.openflowhub.org/display/Beacon.
[39] A. Wang, W. Zhou, B. Godfrey, M. Caesar, Software-defined networks as databases, in: Presented as Part of the Open Networking Summit 2014 (ONS 2014), USENIX, Santa Clara, CA, 2014.
[40] A. Khurshid, W. Zhou, M. Caesar, P.B. Godfrey, Veriflow: verifying network-wide invariants in real time, in: Proceedings of the First Workshop on Hot Topics in Software Defined Networks, HotSDN '12, 2012.

[41] S. Shin, G. Gu, Attacking software-defined networks: a first feasibility study (short paper), in: Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13), 2013.
[42] D. Kreutz, F.M.V. Ramos, P. Verissimo, Towards secure and dependable software-defined networks, in: Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13), 2013.
[43] D.Y. Huang, K. Yocum, A.C. Snoeren, High-fidelity switch models for software-defined network emulation, in: Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN '13, ACM, New York, NY, USA, 2013, pp. 43–48, doi:10.1145/2491185.2491188.
[44] H. Hu, G.-J. Ahn, W. Han, Z. Zhao, Towards a reliable SDN firewall, in: Presented as Part of the Open Networking Summit 2014 (ONS 2014), USENIX, Santa Clara, CA, 2014.

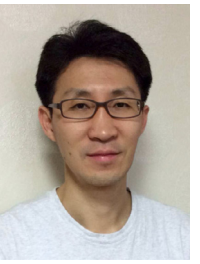
Changhoon Yoon is a Ph.D. student in Graduate School of Information Security at KAIST working with Dr. Seungwon Shin in Network and System Security (NSS) Laboratory. He received his B.S. degree in Computer Engineering from the EECS department of University of Michigan-Ann Arbor. He received his M.S. degree in Information Security from KAIST. His research interests are in the areas of network security, Software-Defined Networking (SDN) security, and malware distribution network.

Taejune Park is a M.S. Student in the Graduate School of Information Security at KAIST working with Dr. Seungwon Shin in NSS Lab. He received his B.S. degree in Computer Engineering from Korea Maritime and Ocean University in Korea. His research interests include security, SDN and software development.

Seungsoo Lee is a M.S. student in the Graduate School of Information Security at KAIST working with Dr. Seungwon Shin in NSS Lab. He received his B.S. degree in Computer Science from Soongsil University in Korea. His research interests include secure and robust SDN controller, and protecting SDN environments from threats.



Heedo Kang is a M.S. student in the Department of Information Security at KAIST working with Dr. Seungwon Shin in NSS Lab. He received his B.S degree in Computer Engineering from Ajou University in Korea. His research interests include interdomain-routing using SDN.



Seungwon Shin is an Assistant Professor of the Graduate School of Information Security at Korea Advanced Institute of Science and Technology (KAIST). He received his Ph.D. degree from the Department of Electrical and Computer Engineering at Texas A&M University. He received his B.S. and M.S. degrees in Electrical Engineering from KAIST in South Korea. His research interests include designing secure SDN architecture/infrastructure, analyzing and detecting botnet, and protecting cloud-computing environments from threats.

Zonghua Zhang is currently an Associate Professor of Institute Mines-Télécom/TELECOM Lille of France. He is also affiliated with CNRS SAMOVAR UMR 5157 as an associate researcher. Previously, he worked as an Expert Researcher at the Information Security Research Center of National Institute of Information and Communications Technology (NICT), Japan from April, 2008 to April, 2010. Even earlier, he spent two years for post-doc research at the University of Waterloo, Canada and INRIA, France after earning his Ph.D. degree in information science from Japan Advanced Institute of Science and Technology (JAIST) in 2006. He has participated in a number of national projects and international collaboration projects, which cover a wide spectrum of security research topics such as anomaly detection, network forensics, root cause analysis, security management, reputation systems, wireless network security, and cryptographic protocols. His current research is focused on large-scale threat analysis, software-defined networking security, and privacy of crowd sensing applications to smart city and eHealthcare.