SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks 2016; **9**:1971–1982
Published online 26 October 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/sec.1369

RESEARCH ARTICLE

Vulnerabilities of network OS and mitigation with state-based permission system

Jiseong Noh, Seunghyeon Lee, Jaehyun Park, Seungwon Shin and Brent Byunghoon Kang*

Graduate School of Information Security, School of Computing, Korea Advanced Institute of Science and Technology, Daejeon, Korea

ABSTRACT

The advancement of software defined networking (SDN) is redefining traditional computer networking architecture. The role of the control plane of SDN is of such importance that SDNs are referred to as network operating systems (OSs). However, the robustness and security of the network OS has been overlooked. In this paper, we report three main issues pertaining to network OSs. First, we identified vulnerabilities that could be exploited by malicious or buggy applications running on network OSs. We also identified four major attack vectors that could undermine network OS operations: denial of service, global data manipulation, control plane poisoning, and system shell execution. Further, it was demonstrated that real-world attacks can be launched on commonly used network OSs without significant effort. Second, we present a method to address the attacks by analyzing network applications running on network OSs to identify their behavioral features, which enabled the extraction of a permission set for each network application. Based on this work, a permission-based malicious network application detector was introduced, which examines the permission set of each application and prevents it from executing without permission. Our system shows almost no performance overhead. Copyright © 2015 John Wiley & Sons, Ltd.

KEYWORDS
network security; software defined networking; network operating system

***Correspondence**
Brent Byunghoon Kang, Graduate School of Information Security, School of Computing, Korea Advanced Institute of Science and Technology, Daejeon, Korea.
E-mail: brentkang@kaist.ac.kr

1. INTRODUCTION

Software defined networking (SDN) is a revolutionary computer networking architecture whose programmability, flexibility, and dynamism serve to distinguish it from legacy computer networks. In general, an SDN with a single centralized controller abstracts underlying network infrastructures (e.g., network routers or switches) to provide network applications with a global view. Considering its similar role to that of traditional operating systems (OSs) (e.g., Linux), namely, the abstraction of system resources to their applications, SDN controllers can be regarded as a network OS [1]. A number of network OSs have been proposed for which there are various network applications available [1–5]. Several research projects [6,7] have employed network OSs, which have even been used in commercial products, such as the Google B4 project and Facebook datacenter [8,9]. In situations such as these, establishing the security, robustness, and reliability of a network OS has become crucial because a network OS would be the main target of attackers who would be able to control a target network simply by compromising a single target (i.e., the network OS).

Considering the vulnerability of a network OS, it would be reasonable to expect that aspects of the security problems would have been addressed in existing studies. However, we discovered that most of the work relating to network OSs was directed at performance issues [10] or distributed architectures [11,12]. In fact, a limited number of studies have attempted to shed some light on the security problems associated with SDN [13]; however, their contributions may not eliminate every possible problem, as their efforts were mainly limited to presenting the possibilities or side effects of attacks [13,14].

The scarcity of studies exposing security (or robustness) issues relating to network OSs prompted our decision to examine these issues. One of these studies [15] introduced issues that could threaten the robustness and security of existing network OSs. Using this paper, we identified possible threat vectors applicable to existing network OSs and categorized them for further analysis. In addition to categorizing the threats, our research also extended to executing attack trials of real-world network OSs. For example, an attack was conducted that was aimed at modifying the internal information of a network OS with the ultimate goal of confusing other critical network applications. Cases are presented in Section 3.

Based on our analysis and categorization of threat vectors, a possible defense mechanism is presented, considering the characteristics of a network OS. Our defense approach consists of an initial analysis of the behavioral operations of existing network applications running on a network OS, following which an attempt is made to generalize these behavioral patterns. These patterns were expected to enable us to extract selected features that would allow the definition of most of the critical operations (or functions) of a network application. Finally, the features that were identified in this manner were implemented to control the operations of a network application with the aim of avoiding unintended and unexpected misbehavior (e.g., crashing the network OS). Using these features, we proceed to define permission-based roles for each network application, which could be used by network administrators as elements for selectively permitting operations, thereby allowing the necessary features for a network application to be specified. Using defined roles for network applications would enable the detection of a network application that behaves abnormally.

The main contributions of our work are categorized as follows.

- Most of the possible attack vectors against a network OS are categorized. This paper describes the implementation of several attack network applications and demonstrates that they are able to crash, confuse, or misuse network OSs without significant effort.
- The operations of a network application are analyzed, and the critical features that could have a serious effect on a network OS are extracted.
- A novel defense mechanism is proposed for defending a network OS against diverse attacks. It is shown that our approach detects (and prevents) attacks by simply defining selected necessary features. This approach shows almost zero overhead in our test cases due to a lightweight design.

2. BACKGROUND

Software defined networking is a new network paradigm that decouples the control plane (deciding how a packet is forwarded to a destination) from the data plane (actually delivering packets based on the decision of the control plane), thereby using software to enable the network to become intelligent. SDN enables a logically centralized network; thus, it is considered a promising architecture capable of addressing current problems in the computer networking area [16]. SDN consists of three main components: (i) a data plane; (ii) a control plane (also known as a network OS or a controller); and (iii) network applications.

Figure 1 shows a simplified SDN network architecture. A brief description is provided of packet delivery from the source to the destination in the network. At first, when a new packet is initiated from the source host (i.e., Host A) and sent to a switch (i), the switch first looks up its flow table to check if a flow rule that can handle this packet is present in its flow table entry. If the switch cannot find a flow rule for handling the packet, it will send a request for a flow rule to the network OS (ii). Next, the network OS delivers this request to the network application, upon which the application (here, we assume that a network application simply forwards all packets to a target host) will generate a flow rule to handle this packet (iii). The network OS will enforce this rule to the switch (iv); and finally, this packet will be forwarded to Host B (v). The OpenFlow [17] protocol is widely used for communication between the switch and the network OS. This protocol is currently the de facto standard protocol for communication between the two entities under an SDN environment.

This example indicates that all components of the data plane in an SDN network should connect to a network OS (or a set of network OSs). Therefore, the network OS will be a centralized control point at which flow rule information is received or from which status information is sent. In this environment, the network OS (and its applications) controls all network flows passing through these network devices. Likewise, the network OS controls the whole

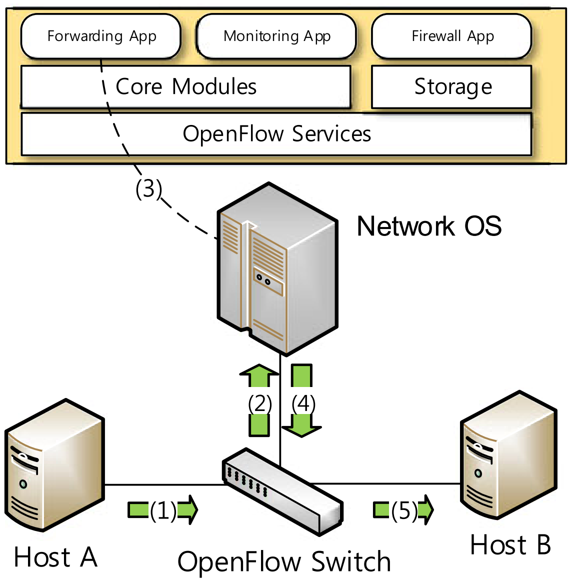


Figure 1. Simple software defined networking operation: how a packet is traversed from source to destination.

network environment; thus, it can be regarded as the brain of a network environment.

3. ATTACKING NETWORK OPERATING SYSTEM

This section presents possible attack scenarios against existing network OSs with real examples.

3.1. Assumption and test environment

The assumption is made that an application, provided by a third-party developer, can be installed on a network OS. We believe that the assumption is feasible because most network OSs highly encourage anyone who has an interest in this area to implement a network application by releasing open APIs [1–5]. Moreover, this trend can be seen in industry, for example, a network application store that has been launched by HP [18].

Our attacks were demonstrated by employing three well-known network OSs: *NOX* version 0.9.2, *FloodLight* version 0.90, and *OpenDayLight* version 1.0. All the tests were conducted in a Mininet version 2.0 [19] environment, which is a popular SDN emulator, and four different types of attacks were tested against these network OSs.

3.2. Attack summary

Table I summarizes the four attack vectors that were found. The denial of service (DoS) attack is either capable of affecting the performance of a network OS or harms the availability of both the network applications and the network OS itself. The internal storage manipulation attack was targeted at all the mentioned types of network OSs, except for OpenDayLight. This form of attack breaks the integrity of internal data and could cause the applications running on the network OS to function in an unintended manner. The data plane poisoning attack affects the data plane by manipulating the flow entries of a switch. The system shell execution allows a network application to execute arbitrary system calls that would be capable of destroying a system. Table I shows these different types of attack and practical example codes executed on FloodLight.

3.3. Denial of service attack

NOX case. There are two ways to perform this attack on NOX: CPU consumption and OpenFlow event delivery prevention. As NOX does not have a resource management scheme, such as CPU and memory usage restriction, running an infinite loop in the application is enough to consume the resources of a network OS. The second approach is to block an OpenFlow event from being sent to other applications. By design, it is the responsibility of each network application to decide whether to keep an existing OpenFlow event or to send it to other applications. When an event handler in one application receives an OpenFlow event, it can decide whether the event should be kept and sent to other applications through its return value. If the return value of the event handler is STOPPED, the event dispatcher in a NOX kernel does not proceed to deliver the OpenFlow event to other applications. An application developer can assign the highest priority for a specific OpenFlow event type in the application to enable it to receive the event first and prevent other applications, including a forwarding application, from receiving the event.

FloodLight case. By design, a FloodLight controller has to wait until an application completes its process. This behavior is very similar to that of NOX. The malicious application is able to loop itself infinitely, upon which FloodLight stops working and waits until the malicious application finishes its work.

OpenDayLight case. An attack on this network OS can be achieved by implementing an application that creates a number of threads. If the number of unnecessary threads is sufficiently large, that is, hundreds, OpenDayLight is slowed down operationally. Table II shows the decrease in performance as the number of threads increases. Eventually, OpenDayLight becomes unresponsive when a malicious application creates more than 500 threads.

3.4. Internal storage manipulation attack

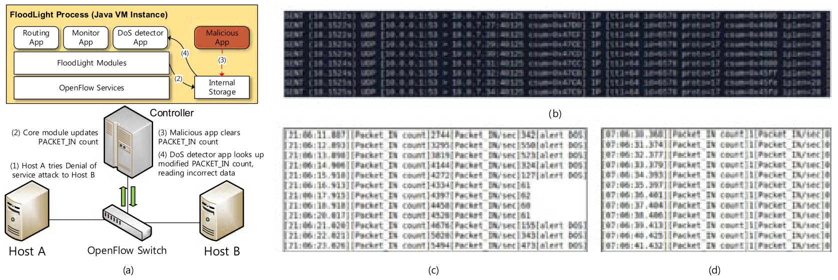
FloodLight case. This type of attack only applies to FloodLight at this time. The attack is performed by allowing a malicious application to access the core database and manipulate its data (i.e., manipulate internal storage). The effectiveness of this type of attack was demonstrated on

Table II. Latency of the first packet forwarding without flow entry in switch according to attack threads.

Thread counts	Initiating packet latency
No thread	0.20 ms
100 threads	0.73 ms
500 threads	Unreachable

Table I. Summary of vulnerable attack vectors on each network operating system.

Attack vectors	NOX	FloodLight	OpenDayLight	Practical example (FloodLight)
Denial of service	O	O	O	return Commands.STOP
Internal storage manipulation	X	O	Unknown	storageSource.executeQuery()
Data plane poisoning	O	O	Partial	sw.clearAllFlowMods()
System shell execution	O	O	O	Runtime.getRuntime().exec()


FloodLight Process (Java VM Instance)
Routing App
Monitor App
DoS detector App
Malicious App
FloodLight Modules
OpenFlow Services
Internal Storage
(4)
(3)
(2)
Controller
(2) Core module updates PACKET_IN count
(1) Host A tries Denial of service attack to Host B
(3) Malicious app clears PACKET_IN count
(4) DoS detector app looks up modified PACKET_IN count, reading incorrect data
Host A
OpenFlow Switch
Host B
(a)
(b)
(c)
(d)

```
SENT (18.1522s) UDP [10.0.0.1:53 > 10.0.7.26:40125 csum=0x47D1] IP [ttl=64 id=6578 proto=17 csum=0x4005 iplen=28 ]
SENT (18.1522s) UDP [10.0.0.1:53 > 10.0.7.27:40125 csum=0x47D0] IP [ttl=64 id=6578 proto=17 csum=0x4004 iplen=28 ]
SENT (18.1523s) UDP [10.0.0.1:53 > 10.0.7.28:40125 csum=0x47CF] IP [ttl=64 id=6578 proto=17 csum=0x4003 iplen=28 ]
SENT (18.1523s) UDP [10.0.0.1:53 > 10.0.7.29:40125 csum=0x47CE] IP [ttl=64 id=6578 proto=17 csum=0x4002 iplen=28 ]
SENT (18.1523s) UDP [10.0.0.1:53 > 10.0.7.30:40125 csum=0x47CD] IP [ttl=64 id=6578 proto=17 csum=0x4001 iplen=28 ]
SENT (18.1524s) UDP [10.0.0.1:53 > 10.0.7.31:40125 csum=0x47CC] IP [ttl=64 id=6578 proto=17 csum=0x4000 iplen=28 ]
SENT (18.1524s) UDP [10.0.0.1:53 > 10.0.7.32:40125 csum=0x47CB] IP [ttl=64 id=6578 proto=17 csum=0x45FF iplen=28 ]
SENT (18.1525s) UDP [10.0.0.1:53 > 10.0.7.33:40125 csum=0x47CA] IP [ttl=64 id=6578 proto=17 csum=0x45fe iplen=28 ]
SENT (18.1525s) UDP [10.0.0.1:53 > 10.0.7.34:40125 csum=0x47C9] IP [ttl=64 id=6578 proto=17 csum=0x45fd iplen=28 ]
```

```
[21:06:11.887]|[Packet_IN count|2744|Packet_IN/sec|342|alert DOS]
[21:06:12.893]|[Packet_IN count|3295|Packet_IN/sec|550|alert DOS]
[21:06:13.898]|[Packet_IN count|3819|Packet_IN/sec|523|alert DOS]
[21:06:14.906]|[Packet_IN count|4144|Packet_IN/sec|324|alert DOS]
[21:06:15.910]|[Packet_IN count|4272|Packet_IN/sec|127|alert DOS]
[21:06:16.913]|[Packet_IN count|4334|Packet_IN/sec|61
[21:06:17.915]|[Packet_IN count|4397|Packet_IN/sec|62
[21:06:18.918]|[Packet_IN count|4458|Packet_IN/sec|60
[21:06:20.017]|[Packet_IN count|4520|Packet_IN/sec|61
[21:06:21.020]|[Packet_IN count|4676|Packet_IN/sec|155|alert DOS]
[21:06:22.021]|[Packet_IN count|5020|Packet_IN/sec|343|alert DOS]
[21:06:23.026]|[Packet_IN count|5494|Packet_IN/sec|473|alert DOS]
```

```
07:06:30.368||Packet_IN count|1|Packet_IN/sec|0
07:06:31.374||Packet_IN count|1|Packet_IN/sec|0
07:06:32.377||Packet_IN count|1|Packet_IN/sec|0
07:06:33.379||Packet_IN count|1|Packet_IN/sec|0
07:06:34.393||Packet_IN count|1|Packet_IN/sec|0
07:06:35.397||Packet_IN count|1|Packet_IN/sec|0
07:06:36.401||Packet_IN count|1|Packet_IN/sec|0
07:06:37.404||Packet_IN count|1|Packet_IN/sec|0
07:06:38.408||Packet_IN count|1|Packet_IN/sec|0
07:06:39.413||Packet_IN count|1|Packet_IN/sec|0
07:06:40.425||Packet_IN count|1|Packet_IN/sec|0
07:06:41.432||Packet_IN count|1|Packet_IN/sec|0
```

Figure 2. An example of data manipulation.

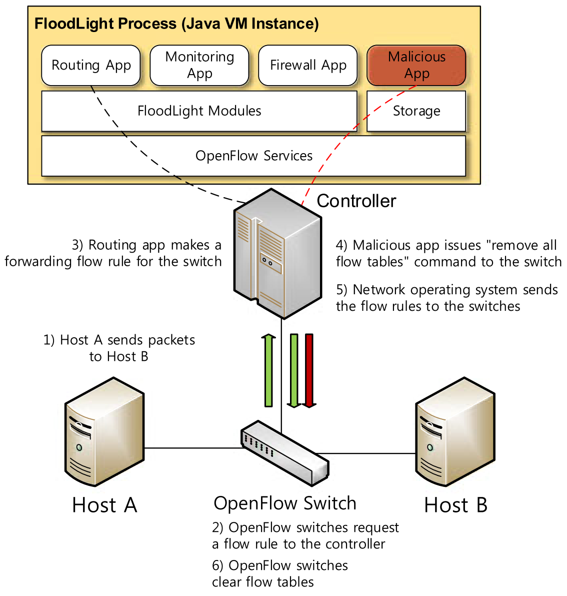

FloodLight Process (Java VM Instance)
Routing App
Monitoring App
Firewall App
Malicious App
FloodLight Modules
Storage
OpenFlow Services
Controller
3) Routing app makes a forwarding flow rule for the switch
4) Malicious app issues "remove all flow tables" command to the switch
5) Network operating system sends the flow rules to the switches
1) Host A sends packets to Host B
Host A
OpenFlow Switch
Host B
2) OpenFlow switches request a flow rule to the controller
6) OpenFlow switches clear flow tables

Figure 3. Example of data plane poisoning attack in FloodLight controller.

FloodLight by configuring the simple test environment shown in Figure 2. The test involved the creation of a simple DoS detector application, which periodically checks the `Packet_in` count value in the internal storage of FloodLight, and it reports alerts if the count value exceeds a predefined threshold value (shown in Figure 2(c)). Then a malicious application is installed (the red box in Figure 2(a)) in order to confuse the detector. This prevents our DoS detector application from detecting an attack, even if a DoS attack takes place (Figure 2(b) and (d)).

3.5. Data plane poisoning attack

FloodLight case. The application of this attack to FloodLight is presented in Figure 3. Here, a malicious application succeeds at flushing out the flow rules from the data plane. According to the OpenFlow protocol, when a new network packet arrives at a data plane that does not contain a flow rule for the arrived packet, the data plane needs to send a flow rule request message to the network OS at the initiation of flow creation, typically once per network flow. However, if the malicious application flushes out all the flow rules, the data plane is forced to send a flow request message every time the network packet arrives at the data plane. This would significantly delay the entire network. As shown in Table III, the round trip time delay measured when the malicious application was in effect was 100 times more than without the malicious application.

Table III. Average round trip time before and after the data plane poisoning attack.

	NOX	FloodLight	OpenDayLight
Normal	0.05 ms	0.05 ms	0.08 ms
Compromised	2.00 ms	5.00 ms	0.11 ms
Delay rate	3900%	9900%	37.5%

In addition, the way in which the transmission of a critical flow rule impacts the performance was demonstrated by comparing the traverse time, as shown in Table III. When a malicious application periodically transmits a delete flow rule to a switch, the overall performance is degraded significantly. Since the malicious application deletes all the flow rules, the switch would send a query to the control plane for every packet. Sending a critical flow rule would be able to cause security holes, such as bypassing the firewall, intrusion detection system, and other network security devices.

NOX case. Similar to FloodLight, NOX does not have a restriction mechanism for sending an arbitrary flow rule to the data plane, regardless of which application sends a flow rule intended for a developer or an adversary. The effect of this kind of attack is clearly presented in Table III.

OpenDayLight case. Since OpenDayLight does not check the authenticity of a flow rule sent to the data plane, a malicious application was used to perform the data plane poisoning attack as well. As shown in Table III, however, the latency of the round trip time does not increase significantly compared with the cases of NOX and FloodLight although the malicious application successfully affected the data plane. This result shows that OpenDayLight may be resilient to the data plane poisoning attack.

3.6. System shell execution attack

During this type of attack, a malicious application can create a number of new processes to consume the resources of the host computer running FloodLight. As an operation of this nature would not be restricted by a network OS, the application would be able to conduct various critical operations that could alter the normal operations of FloodLight, NOX, or OpenDayLight. None of the network OSs were found to provide any means for restricting the invocation of a sensitive system call (e.g., a system exit function), and thus, all network OSs would be prone to this type of attack.

4. DEFENDING NETWORK OPERATING SYSTEM FROM ATTACKS

4.1. Overview

In the previous section, we presented real-world examples of situations in which network OSs could be attacked. Network users naturally consider a defense system that protects a network OS from these attack trials. However, a very limited number of studies have addressed this problem. For example, Monaco *et al.* [20] suggested a method that employs generic OS features to increase the robustness of the network OS, and Wen *et al.* [14] proposed a system that prohibits an application from invoking system calls without permission. Another study [15] attempted to create a robust and secure network OS by completely redesigning the architecture of the network OS. The Rosemary network OS has various built-in protection mechanisms to protect network applications and the network OS itself. Although they conducted pioneering research work in this area, they did not address all possible problems; hence, more research work is required. One aspect to consider when adding security features to a network OS is ease of deployability, in other words, minimizing the need to change existing network equipment. This motivated us to design a system capable of defending a network OS against diverse attack situations (including the attacks we are proposing in this paper). To this end, we propose a permission-based defense approach for the network OS. This approach is based on Android security features, which have various permission lists named manifest. However, it is difficult to deploy an Android permission system on a network OS as the Android OS is designed for mobile devices. Because of the difference between the features of SDN applications and those of Android applications, deploying such a permission system on a network OS would require significant effort. It would involve extracting the features of SDN applications and designing a new permission system that is compatible with the SDN applications. This has motivated us to design a system capable of defending a network OS from diverse attack cases (including the attacks we have proposed). To this end, we propose a *permission-based defending approach* for network OSs. This approach first generalizes the behavior of network applications running on a network OS to extract common behavioral features. Based on the extracted features, our proposed system provides a way to define allowed and disallowed behaviors for a network application. Considering the generalized behavior of the SDN applications, we suggest an SDN-specific permission system capable of restricting the possible actions of each SDN application.

4.2. Behavior of network applications

Various network applications running on multiple network OSs[†] were analyzed to define their behavior. Although legacy applications (e.g., MS Word and Notepad) could also be considered network applications, the latter applications are likely to show some common behavioral patterns because they follow the OpenFlow protocol and respond according to pre-specified behavioral patterns. Our analysis revealed two important common features that distinguish a network application from a legacy application. First, input locations are limited. Unlike a general application that has almost unlimited input types and values, a network application receives inputs from limited locations. Second, the behavior of network applications is relatively restricted compared with that of a legacy application. A general application may be capable of performing actions that are quite diverse (e.g., copy, read, write, and remove) for a particular event, which requires the program to be complicated. However, a network application is likely to have a limited behavioral set for such an event, such as creating a flow rule and requesting a core module for sending the rule back to the data plane. We believe it should be possible to use these two permission sets in defining the behavior of a network application and thus that they would be able to assist us to define permissions for a network application. Figure 4 shows the difference between the features of network and generic applications. It is clear that generic applications receive various user inputs and produce results according to these inputs. On the other hand, a network application receives an OpenFlow request message from the data plane in reply to which it returns a response message and stores the process information to internal storage.

4.3. Generalizing behavior of network applications

The analysis presented in Section 4.2 indicated the main role of network applications to be the processing of requests from the data plane, a finding on which we based our investigation of publically available SDN applications from various network OSs such as NOX, FloodLight, and OpenDaylight. As a result of this analysis, we defined the term `State` of a network application. A specific OpenFlow message from the data plane is defined as a `State` of an SDN application.

[†] In our analysis, we include network applications distributed with FloodLight, POX, NOX, and OpenDayLight.

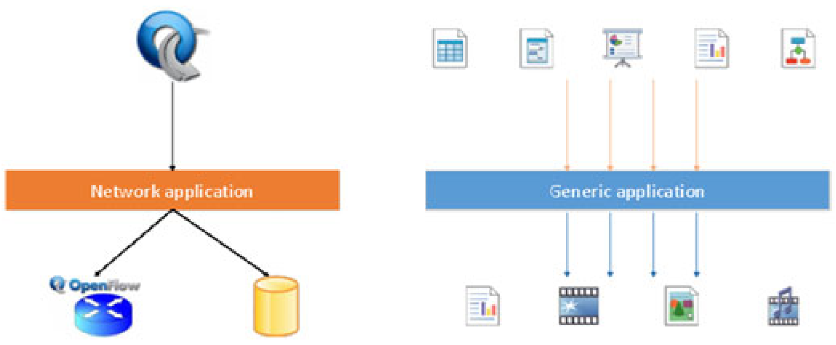


Figure 4. Difference between network applications and generic application.

4.3.1. OpenFlow state.

In general, the life cycle of a network application is quite simple. It waits for a request from the data plane, receives the request from it, processes the request, and sends the result back to the data plane.

According to Openflow specification version 1.1, there are three different OpenFlow requests from the data plane: `Packet_in`, `Flow_removed`, and `Port_status`, each of which is processed differently by the network application.

These three different types of requests are defined as `States`. The `State` is used as the unit to which the permission details can be applied. That is, the permission set of an application can be changed on each different `State`. For example, the role of a Layer 2 Learning Switch application on the FloodLight network OS is to add or remove flow entries of opposite switch when a flow is created or removed in a switch. When a `Packet_in` request has occurred, this application generates either a `Packet_out` or a `Flowmod_add` message, which sends back to the corresponding opposite switch. It generates and sends a `Flowmod_delete` message when a `Flow_removed` request has occurred. Without `State` information, this application will always be allowed to send `Packet_out`, `Flowmod_add`, and `Flowmod_delete` messages in any situation. On the other hand, the application can be required to obtain permission to send a message if it has `State` information. This state-related permission set enables the system administrator to have an increased awareness of the activities of each application. Based on the OpenFlow specification version 1.1 and our analysis of SDN applications, we found that a network application can have up to five states. The `States` are listed as follows.

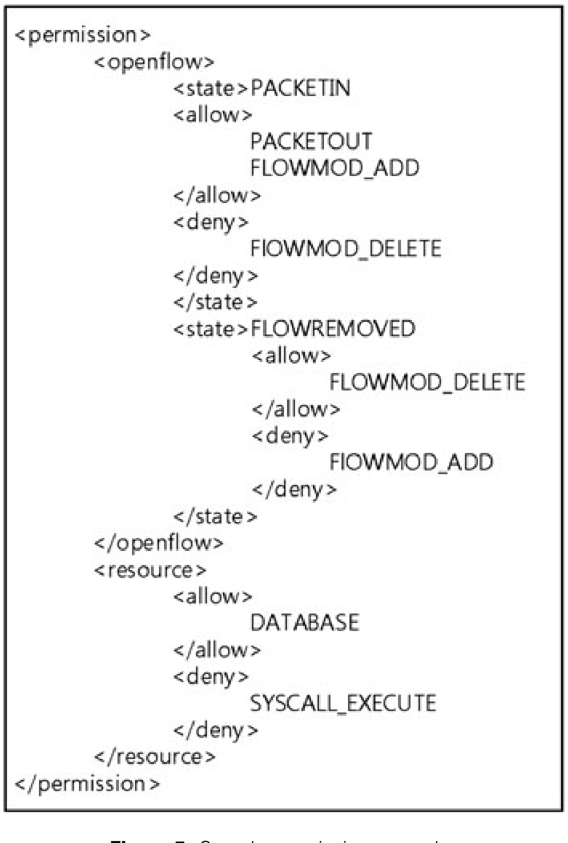
- Initial `State`. When the controller starts, applications are in the Initial `State`, in which applications are initialized and configured.
- Ready `State`. After initialization, applications are in the Ready `State`, that is, they are ready to receive a new event. This is the basic `State` in which all applications are positioned before receiving a request from the data plane.
- `PACKETIN State`. When the controller receives a `Packet_in` OpenFlow message and forwards the message to a network application, the receiving application changes its `State` to `PACKETIN State`.
- `FLOWREM State`. When the controller receives a `Flow_removed` OpenFlow message, an application with permission to process it will change its `State` to `FLOWREM State`.
- `PORTSTATUS State`. If the controller receives a `Port_status` OpenFlow message, an application with permission to process it will change its `State` to `PORTSTATUS State`.

4.3.2. Permission sets for SDN applications.

Current network OSs do not impose explicit restrictions on SDN applications. This allows buggy or malicious applications to run arbitrary commands, which may be harmful to the network OS, as described in Section 3. By adding permission rules to SDN applications, the network administrator is able to limit application permissions to impede malicious activities. This permission model enables each network application to have the least privilege principle, which improves the security of the network OS. Our model is similar to the Android permission model, which supports a scalable permission set. In our permission policy, there are two main permission sets on SDN applications. One is the OpenFlow message permission set, in which each type of OpenFlow message is listed as a permission rule. The definition of the OpenFlow `States` in Section 4.3.1 enables us to create permission sets efficiently. For example, it would be possible to allow a forwarding application to only receive `Packet_in` OpenFlow messages by restricting its states to Initial, Ready, and `PACKETIN States` only. The other permission set is the network OS resource permission set. Current network OS architecture permits SDN applications to access network OS resources without any restriction. As demonstrated in Section 3, malicious SDN applications can modify data on internal storage for DoS or they could even open a backdoor to allow an unauthorized entity to access the network. In general, SDN applications share network OS resources, which necessitate a tracking process to establish which entity has access to a particular shared resource and to identify which action the entity performs on the shared resource. Tracking requires the implementation of authentication. Authentication, coupled with a network OS resource permission system, would effectively defend the system against malware or malicious attacks. We created two network OS resource permission sets: (i) permission on accessing internal storage and (ii) permission on system call.

4.4. Detection strategy

In the previous subsection, we described the states and permissions that may be assigned to a network application. The permission set of a network application consists of an OpenFlow message and a network OS resource at running time. These two features of the network application can be combined and structured in a format similar to XML. The format is illustrated in Figure 5. The OpenFlow permission set is highly dependent on a state. That is, the permission set can be changed for different OpenFlow `States`. For example, an application that has two OpenFlow states may have a different permission set for each of these `States`. Figure 5 shows the permission format. The two main permission sets are OpenFlow and System resource, and these two sets are independent from one another. OpenFlow permission is required to start from a state tag to set the permission along with the corresponding state. Some permission lists have subpermission rules,



```
<permission>
        <openflow>
                <state>PACKETIN
                <allow>
                        PACKETOUT
                        FLOWMOD_ADD
                </allow>
                <deny>
                        FIOWMOD_DELETE
                </deny>
                </state>
                <state>FLOWREMOVED
                        <allow>
                                FLOWMOD_DELETE
                        </allow>
                        <deny>
                                FIOWMOD_ADD
                        </deny>
                </state>
        </openflow>
        <resource>
                <allow>
                        DATABASE
                </allow>
                <deny>
                        SYSCALL_EXECUTE
                </deny>
        </resource>
</permission>
```

Figure 5. Sample permission example.

Table IV. Implemented permission sets.

Type	Permission name	Description
OpenFlow	PACKETOUT	Send Packet_out
	FLOWMOD_ADD	Send Flowmod_add
	FLOWMOD_DELETE	Send Flowmod_delete
	FLOWMOD_MODIFY	Send Flowmod_modify
	STATSREQUEST	Send stats_request
System	DATABASE	Access internal storage
	SYSTEMCALL	Execute system call

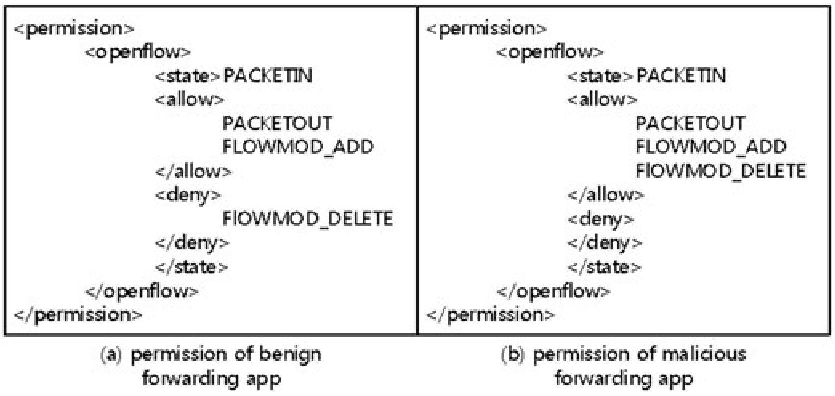
such as FLOWMOD_ADD, FLOWMOD_DELETE, and SYSCALL_EXECUTE. All the rules in the permission sets are listed in Table IV. For the unspecified permission lists, the administrator can choose allow all, deny all, or allow with alert. With this permission set, we are able to see how many permissions an application requires at certain `State`. Figure 6 shows the required permission difference between the benign and malicious forwarding applications.

5. IMPLEMENTATION AND EVALUATION

The performance of our detection strategy was validated by implementing a prototype of a permission-based detection system on FloodLight network OS version 0.90. Our implementation was found to effectively detect most threats that were identified and described in Section 3 with little overhead. The remainder of this section describes the implementation and evaluation of our prototype.

5.1. Implementation

A prototype of a permission-based detection system was developed on FloodLight with the aim of protecting network OSs against the threats mentioned in Table I. FloodLight version 0.90, running on Ubuntu Linux 13.04, was used for this purpose. The SDN configuration was performed within a Mininet environment [19]. The main challenge in implementing the detection system was to determine how to track all the permission sets in real time, and this was carried out by assuming that network application developers would use our wrapper functions in a way similar to that of Fresco [21]. Our wrapper functions enable the network OS to track the permission set of each application in real time, which was accomplished by creating a wrapper function that performs requested functionalities on behalf of the network application. The wrapper function checks the permission through the permission set of each application before performing the requested function. Based on the wrapper function, a prototype of the permission-based malicious application detection system was built. The prototype operates as follows. First, on application loading time, the network OS parses the permission file that contains the permission set of each of the applications. Second, when an application tries to perform



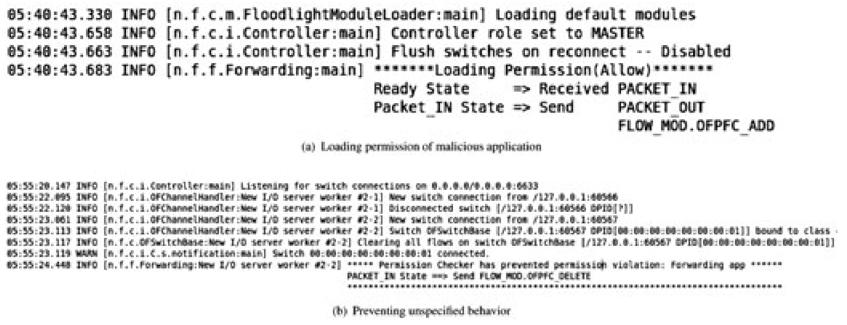
```
<permission>
        <openflow>
                <state> PACKETIN
                <allow>
                        PACKETOUT
                        FLOWMOD_ADD
                </allow>
                <deny>
                        FlOWMOD_DELETE
                </deny>
                </state>
        </openflow>
</permission>
```

(a) permission of benign forwarding app

```
<permission>
        <openflow>
                <state> PACKETIN
                <allow>
                        PACKETOUT
                        FLOWMOD_ADD
                        FlOWMOD_DELETE
                </allow>
                <deny>
                </deny>
                </state>
        </openflow>
</permission>
```

(b) permission of malicious forwarding app

Figure 6. Different permissions of benign and malicious applications.



```
05:40:43.330 INFO [n.f.c.m.FloodlightModuleLoader:main] Loading default modules
05:40:43.658 INFO [n.f.c.i.Controller:main] Controller role set to MASTER
05:40:43.663 INFO [n.f.c.i.Controller:main] Flush switches on reconnect -- Disabled
05:40:43.683 INFO [n.f.f.Forwarding:main] *******Loading Permission(Allow)*******
                                          Ready State     => Received PACKET_IN
                                          Packet_IN State => Send     PACKET_OUT
                                                                      FLOW_MOD.OFPFC_ADD
```

(a) Loading permission of malicious application

```
05:55:20.147 INFO [n.f.c.i.Controller:main] Listening for switch connections on 0.0.0.0/0.0.0.0:6633
05:55:22.095 INFO [n.f.c.i.OFChannelHandler:New I/O server worker #2-1] New switch connection from /127.0.0.1:60566
05:55:22.120 INFO [n.f.c.i.OFChannelHandler:New I/O server worker #2-1] Disconnected switch [/127.0.0.1:60566 DPID[?]]
05:55:23.061 INFO [n.f.c.i.OFChannelHandler:New I/O server worker #2-2] New switch connection from /127.0.0.1:60567
05:55:23.113 INFO [n.f.c.i.OFChannelHandler:New I/O server worker #2-2] Switch OFSwitchBase [/127.0.0.1:60567 DPID[00:00:00:00:00:00:00:01]] bound to class
05:55:23.117 INFO [n.f.c.OFSwitchBase:New I/O server worker #2-2] Clearing all flows on switch OFSwitchBase [/127.0.0.1:60567 DPID[00:00:00:00:00:00:00:01]]
05:55:23.119 WARN [n.f.c.i.C.s.notification:main] Switch 00:00:00:00:00:00:00:01 connected.
05:55:24.448 INFO [n.f.f.Forwarding:New I/O server worker #2-2] ***** Permission Checker has prevented permission violation: Forwarding app ******
                                                                PACKET_IN State ==> Send FLOW_MOD.OFPFC_DELETE
                                                                ******************************************************************************
```

(b) Preventing unspecified behavior

Figure 7. Preventing malicious behavior that disguises firewall application. (a) Loading permission of malicious application. (b) Preventing unspecified behavior.

an action, the wrapper function verifies whether the action is allowed, based on the permission set of the application. Finally, if an application does not have permission to perform the action, our system notifies the administrator of the action of the application and ignores the requested action.

5.2. Evaluation

5.2.1. Effectiveness of attacks.

Our results showed that off-the-shelf network OSs, such as NOX, FloodLight, and OpenDayLight, are all highly vulnerable to the four kinds of attacks (identified in Section 3) by malicious network applications.

5.2.2. Evaluation of the permission system.

The ability of our detection system to effectively prevent the attacks described in Section 3 was demonstrated by creating an example scenario of an attack based on a system call execution attack. We have installed a maliciously crafted network application that launches a system shell command (e.g., launching/bin/bash). This malicious application disguises itself as a firewall application and uses the same permission set as that of a benign firewall application. Figure 7(a) shows the permission set in use when the application is being loaded. When the malicious application attempted a `System shell execution` attack, our detection system successfully prevented the application from executing the system call because the application does not have a privilege to perform the call on its permission set. Figure 7(b) shows that our permission-based detection system prototype is capable of successfully detecting the malicious behavior.

The overhead consumption of our detection system was assessed by measuring the performance under two scenarios, namely, with and without the permission detection system. For both scenarios, the start and end times were recorded. Our performance tests were conducted with a virtual machine and Mininet environment. The host test system consisted of Intel i7-4850HQ and 16 GB of memory running on OS X 10.10 (Yosemite) and VMware Fusion 7. The virtual machine was configured with a two-core CPU and 2.5 GB memory running on Ubuntu Linux 14.04 64 bit.

Table V presents the results of our performance test, which shows that the overhead is within the margin

Table V. Performance evaluation of permission enforced FloodLight (100 runs average).

Without permission	With permission	Overhead
52.3μs	52.8μs	0.5 μs (0.9%)

of accuracy Java `System.nanoTime()`. This result shows that our permission-based detection system generates little overhead, for the reason that the number of permission sets used by our system is designed to behave in a coarse-grained manner. The number of possible permissions is equal to the number of states times the number of possible permissions of each state, which may be considered as involving no overhead.

6. DISCUSSION

The prevention of DoS-type attacks using our permission framework is currently beyond the scope of our work. Although our permission framework may be extended to include the allowed rates and frequencies of contacts, the permission framework generally does not address the exorbitant usage of permitted paths or executions. A number of outstanding problems relating to solving the problems identified and listed in Table I still remain. These limitations mainly exist because of an insufficient number of sample network applications, which also proved problematic in other respects. Our attack scenarios were defended by producing an anomaly detection system based on our analysis of network applications through permissions. However, our analysis was complicated by the limited number of available network applications that could be used to classify benign and malicious behaviors. In the near future, we can expect increasing activity in the market for network OS applications, resulting in a multitude of application samples. Once this occurs, we would be able to expand our research on anomaly detection by using some machine learning techniques, such as support vector machine (SVM). Despite these limitations, the attacks we identified and the permission system we developed relate to real-world network OS. It is expected that we would be able to solve the aforementioned problems during our ongoing research efforts.

This permission system could be applied to examine SDN applications, especially for public application stores. For example, the administrators of the HP SDN application store could examine a submitted SDN application by comparing the application description with the application permission, verifying whether the application has permissions that are not written in the description. Because over privileged applications may have potential malicious codes, administrators could refuse to publish these applications or ask the developers to fix the issues.

7. RELATED WORK

Several pioneering studies have been conducted to investigate security issues relating to SDN [13]. Possible threat vectors in SDN were studied [22], possible security problems presented by the OpenFlow protocol were tracked [23], and other possible attack scenarios were discovered [24,25]. However, our work differs from these studies in that it is the first attempt to illustrate real-world attack scenarios involving the subversion of a network OS. Previous work focused on the data plane as being the most vulnerable to external attacks. Typically, adverse behavior does not physically access the control plane. While the industry was still considering opening an SDN application store, adversaries would have had a gateway for hacking network OSs with a maliciously crafted network application. However, now that an SDN application store [18] has been launched, adversaries have a way to access the control plane by creating maliciously crafted network applications and distributing them on the SDN application store.

To prevent unintended results, several researchers focused on SDN security features. FortNOX [26] prevents rule conflicts among applications in real time. SEFloodLight [27] is an extension and improvement of the FortNOX system implemented in FloodLight. NICE [28] is a tool for automating OpenFlow application testing to find bugs and evaluate the reliability of a network application. YANC [20] adopts UNIX permission separation in modern OSs to a network application through a file system concept.

Recently, research concerning the security of an SDN controller [29–31] was conducted. This work suggests a growing interest in security issues relating to network OSs.

There have been many efforts to investigate permission-based security enhancement. Specifically, research involving the use of a permission set to provide improved access control on Android [32] OSs is one of the security issues currently receiving the most attention [33–36]. However, no work has yet been performed in connection with permission set systems on network OSs. To the best of our knowledge, this is the first work suggesting the use of a proper permission set for each network application based on our analysis of the network application behavior in the context of SDN. To date, few studies have dealt with permission-based authentication systems for SDN [14,37]. PermOF [14] proposes the use of 18 permission sets under four distinct categories (Read, Notification, Write, and System). The former three categories are OpenFlow-related permissions, whereas the remaining category controls the access to the local resources provided by a network OS, such as network access or file system access. OperationCheckpoint [37] adopts the part of the permission set of PermOF, which was implemented on the FloodLight controller. However, this permission set is too coarse grained and does not enable network application users to distinguish malicious applications from benign applications. In contrast to their approach, our permission system has five states for each of the permissions, which help application users to determine when the application uses each OpenFlow protocol.

Our previous work [15] proposed a security-enhanced network OS based on the micro-NOS concept that spawns network applications independently. Following the micro-NOS concept, the network OS, named, Rosemary Kernel, and each network application are launched in separate processes at different levels of security (kernel mode and user mode, inspired by exokernel [38]). Thus, a network application cannot invade network OS resources, such as network topology information in its memory, without proper permissions. Moreover, a network application is monitored by the Resource Manager, which inspects whether network applications exceed their own capabilities in terms of memory, CPU, file descriptor, and network usage as accounted in their permission assessment. Although Rosemary effectively prevents DoS and internal storage manipulation attacks, it is not designed to prevent well-crafted malicious applications from poisoning the data plane. As mentioned earlier, because of the state-based permission system, most malicious behavior can be filtered in the application install phase.

8. CONCLUSION

We demonstrated that network OSs currently in use are vulnerable and can be exploited through various attack vectors as shown by the exploitation examples. Most existing network OSs that were analyzed do not include security concepts that may prevent attacks such as DoS, internal storage manipulation, data plane poisoning, and system shell execution. An analysis of the behavior of network applications enabled us to extract the unique features of those applications to model the normal operation of each application role. A table containing the permission set was created for each application, and behavioral patterns were clustered in an attempt to group user applications according to their functionality. Moreover, we suggested an approach for the classification of core and user applications. A prototype abnormal behavior network application detector was implemented on FloodLight using the aforementioned features we identified. Future work aims to adapt existing state-of-the-art work, such as static analysis and dynamic analysis.

ACKNOWLEDGEMENTS

This research was supported by the Ministry of Science, ICT and Future Planning (MSIP), Korea, under the BrainScouting-Program (H7106-14-1011) supervised by the Institute for Information and Communication Promotion (IITP). This work was also sponsored by the Agency for Defense Development (ADD) under grant no. UD140002ED. This work was also supported by the ICT R&D program of MSIP/IITP (R-20150223-000167, Development of High Reliable Communications and Security SW for Various Unmanned Vehicles). This work was also supported by grant no. K-15-L01-C04-S01 from the Korea Institute of Science and Technology Information and was supported by the ICT R&D program of MSIP/IITP (2014-044-072-003, Development of Cyber Quarantine System using SDN Techniques).

REFERENCES

1. Gude N, Koponen T, Pettit J, *et al.* NOX: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review* 2008; **38**(3): 105–110.
2. Mccauley J. POX: a Python-based OpenFlow controller. Available from: http://www.noxrepo.org/pox/about-pox/ [Accessed on 2 October 2015].
3. Big S. Floodlight openflow controller. Available from: http://www.projectfloodlight.org/floodlight/ [Accessed on 2 October 2015].
4. OpenFlowHub. BEACON. Available from: http://www.openflowhub.org/display/Beacon [Accessed on 2 October 2015].
5. Foundation L. OpenDaylight. Available from: https://www.opendaylight.org/ [2 October 2015].
6. Khurshid A, Zhou W, Caesar M, Godfrey P. Veriflow: verifying network-wide invariants in real time. *ACM SIGCOMM Computer Communication Review* 2012; **42**(4): 467–472.
7. Braga R, Mota E, Passito A. Lightweight ddos flooding attack detection using nox/openflow. *2010 IEEE 35th Conference on Local Computer Networks (LCN)*, Denver, CO, USA, 2010; 408–415.
8. Jain S, Kumar A, Mandal S, *et al.* B4: experience with a globally-deployed software defined WAN. *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM.* ACM, Hong Kong, China, 2013; 3–14.
9. Heiliger J. Building efficient data centers with the open compute project, 2011. Available from: https://www.facebook.com/notes/facebook-engineering/building-efficient-data-centers-with-the-open-compute-project/10150144039563920 [Accessed on 2 October 2015].
10. Dixit A, Hao F, Mukherjee S, Lakshman TV, Kompella R. Towards an elastic distributed SDN controller. *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking.* ACM, Hong Kong, China, 2013; 7–12.
11. Koponen T, Casado M, Gude N, *et al.* Onix: a distributed control platform for large-scale production networks. *Proceedings of the 9th USENIX conference on Operating systems design and implementation.* Vancouver, BC, Canada, 2010; 1–6.
12. Tootoonchian A, Ganjali Y. Hyperflow: a distributed control plane for openflow. *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking.* USENIX Association, San Jose, CA, USA, 2010; 3–3.
13. Kreutz D, Ramos F, Verissimo P. Towards secure and dependable software-defined networks. *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking.* ACM, Hong Kong, China, 2013; 55–60.
14. Wen X, Chen Y, Hu C, Shi C, Wang Y. Towards a secure controller platform for openflow applications. *Proceedings of the Second ACM SIG-*

COMM Workshop on Hot Topics in Software Defined Networking. ACM, Hong Kong, China, 2013; 171–172.

15. Shin S, Song Y, Lee T, Lee S, Chung J, Porras P, Yegneswaran V, Noh J, Kang BB. Rosemary: a robust, secure, and high-performance network operating system. *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security.* ACM, Scottsdale, AZ, USA, 2014; 78–89.
16. Melvin M. Network Computing. Available from: http://www.networkcomputing.com/networking/can-sdn-adoption-solve-real-world-problems/a/d-id/1005791 [Accessed on 2 October 2015].
17. McKeown N, Anderson T, Balakrishnan H, Parulkar G, Peterson L, Rexford J, Shenker S, Turner J. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* 2008; **38**(2): 69–74.
18. HP. SDN App Store. Available from: http://h17007.www1.hp.com/us/en/networking/solutions/technology/sdn/devcenter/index.aspx#tab=TAB1 [Accessed on October 2 2015].
19. Mininet. Rapid prototyping for software defined networks. Available from: http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet/ [Accessed on 2 October 2015].
20. Monaco M, Michel O, Keller E. Applying operating system principles to sdn controller design. *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks.* ACM, College Park, MD, USA, 2013; 2.
21. Shin S, Porras P, Yegneswaran V, Fong M, Gu G, Tyson M. Fresco: modular composable security services for software-defined networks. *Proceedings of 20th Annual Network & Distributed System Security Symposium.* San Diego, CA, USA, 2013.
22. Shin S, Gu G. Attacking software-defined networks: a first feasibility study. *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking.* ACM, Hong Kong, China, 2013; 165–166.
23. Klöti R. Openflow: a security analysis. *8th Workshop on Secure Network Protocols (NPSEC 2013)*, Göttingen, Germany, 2013; 1–6.
24. Scott-Hayward S, O'Callaghan G, Sezer S. Sdn security: a survey. *2013 IEEE SDN for Future Networks and Services (SDN4FNS).* IEEE, Trento, Italy, 2013; 1–7.
25. Shalimov A, Zuikov D, Zimarina D, Pashkov V, Smeliansky R. Advanced study of SDN/openflow controllers. *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia.* ACM, Moscow, Russia, 2013; 1.
26. Porras P, Shin S, Yegneswaran V, Fong M, Tyson M, Gu G. A security enforcement kernel for OpenFlow networks. *Proceedings of the first workshop on Hot topics in Software Defined Networking.* Helsinki, Finland, 2012; 121–126.
27. OpenFlowSec.org. SEFloodlight. Available from: http://www.openflowsec.org/Home.html [Accessed on 2 October 2015].
28. Canini M, Venzano D, Peresini P, Kostic D, Rexford J. A NICE way to test OpenFlow applications. *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation.* San Jose, CA, USA, 2012; 127–140.
29. McGillicuddy S. SDN security issues: how secure is the SDN stack? Available from: http://searchsdn.techtarget.com/news/2240214438/SDN-security-issues-How-secure-is-the-SDN-stack [Accessed on 2 October 2015].
30. Weinberg N. Is SDN your next security nightmare? Available from: http://www.networkworld.com/news/2014/022814-rsa-sdn-security-279298.html [Accessed on 2 October 2015].
31. Prince B. Beware SDN security risks, experts warn. Available from: www.networkcomputing.com/next-generation-data-center/news/networking/beware-sdn-security-risks-experts-warn/240166081 [Accessed on 2 October 2015].
32. Android Open Source Project. What is Android? Introduction to Android. Available from: http://developer.android.com/guide/index.html [Accessed on 2 October 2015.
33. Barrera D, Kayacik HG, van Oorschot PC, Somayaji A. A methodology for empirical analysis of permission-based security models and its application to android. *Proceedings of the 17th ACM Conference on Computer and Communications Security.* ACM, Chicago, IL, USA, 2010; 73–84.
34. Felt AP, Chin E, Hanna S, Song D, Wagner D. Android permissions demystified. *Proceedings of the 18th ACM Conference on Computer and Communications Security.* ACM, Chicago, IL, USA, 2011; 627–638.
35. Au KWY, Zhou YF, Huang Z, Lie D. Pscout: analyzing the android permission specification. *Proceedings of the 2012 ACM Conference on Computer and Communications Security.* ACM, Raleigh, NC, USA, 2012; 217–228.
36. Smalley S, Craig R. Security enhanced (SE) android: bringing flexible MAC to Android. *Proceedings of 20th Annual Network & Distributed System Security Symposium*, San Diego, CA, USA, 2013.

37. Scott-Hayward S, Kane C, Sezer S. Operationcheckpoint: SDN application control. *2014 IEEE 22nd International Conference on Network Protocols (ICNP).* IEEE, The Research Triangle, NC, USA, 2014; 618–623.

38. Engler DR, Kaashoek MF, O'Toole JW. Exokernel: An Operating System Architecture for Application-level Resource Management. *Proceedings of the fifteenth ACM symposium on Operating systems principles.* Copper Mountain, CO, USA, 1995; 251–266.