SDN Malware: Problems of Current Protection Systems and Potential Countermeasures

Christian Röpke[1]

Abstract: Software-Defined Networking (SDN) is an emerging topic and securing its data and control plane is of great importance. The main goal of malicious SDN applications would be to compromise the SDN controller which is responsible for managing the SDN-based network. In this paper, we discuss two existent mechanisms aiming at protecting aforementioned planes: (i) sandboxing of SDN applications and (ii) checking for network invariants. We argue that both fail in case of sophisticated malicious SDN applications such as a SDN rootkit. To fill the corresponding security gaps, we propose two security improvements. The first one aims at protecting the control plane by isolating SDN applications by means of virtualization techniques. Compared to recent efforts, we thereby allow a more stringent separation of malicious SDN applications. The goal of the second proposal is to allow policy checking mechanisms to run independently from SDN controllers while minimizing hardware costs. Thereby, we improve SDN security while taking into account that correct functioning of policy checking can be manipulated by a compromised SDN controller.

Keywords: Software-defined networking, malicious SDN applications, SDN controller security.

1 Introduction

SDN is a new trend in the field of computer networking. It started in 2008 with OpenFlow [Mc08] which is a protocol for managing programmable switches by third-party control software. Since then, large companies such as HP, IBM and Brocade have introduced SDN-enabled switches [Opb] and researchers as well as enterprises have developed SDN control software (also known as *SDN controller* or *network operating system*) [Gu08, Fl, Po15, Sh14, Opc, ON, SDa, SDb, SDc]. Furthermore, multiple *SDN applications* are available, *e. g.*, in the industry's first SDN app store which is run by HP [HP13]. At the same time, a new research area has emerged including the use of SDN to enhance network security [Al15, Sh13a] as well as securing the SDN architecture [Po12, Sh13b, Sh14, RH15a, RH]. Nowadays, Gartner lists SDN as one of the top 10 strategic technology trends [Ga14a] and predicts that: "By the end of 2016, more than 10,000 enterprises worldwide will have deployed SDN in their networks" [Ga14b]. As SDN has increasingly attracted attention in both academia and industry, SDN has evolved from a research project to an emerging topic.

Technically, SDN decouples the control software, which decides where to forward network packets (*control plane*), from the forwarding hardware, which solely forwards network packets according to the made decision (*data plane*). As illustrated in Fig. 1, the

[1] Ruhr-University Bochum, Chair for System Security, Universitätsstrasse 150, 44801 Bochum, Germany, christian.roepke@rub.de

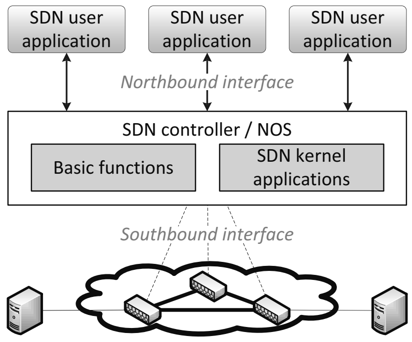


Fig. 1: SDN Architecture

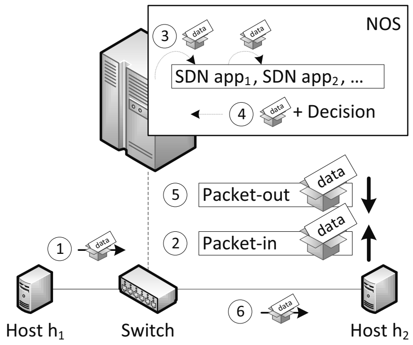


Fig. 2: OpenFlow Reactive Programming

decoupled control software is typically called *SDN controller* or *Network Operating System* (NOS). It is a logically centralized system with the main purpose of implementing network intelligence by programming the network. A SDN-based network thereby consists of programmable network devices and the interface between such devices and the NOS is typically called *southbound interface*. A popular southbound protocol is the aforementioned OpenFlow protocol which allows programming of network devices as well as requesting the network state. On top of a NOS, so called *SDN applications* implement network functionality as required by a network operator. In addition to the southbound interface, the interface between SDN controllers and SDN applications is called *northbound interface*. Inside a NOS, basic functionality is often extended by components typically also called as SDN applications. To distinguish between these two types of SDN applications, we call the ones using the northbound interface *SDN user applications* and the other ones *SDN kernel applications*. We thereby follow the naming convention of previous studies [Sh14, RH15b, RH].

The big benefit of SDN is that network operators can implement and adapt network functionality such as switching or routing independently from the network hardware vendor. For example, if a switch does not provide a feature which is needed or desirable by a network operator, such a feature can be implemented easily by using SDN to program network devices accordingly. In fact, industry is currently adopting SDN (17% of firms surveyed by Gartner already use SDN in production [Ga14b]) and large companies such as Google already benefit from OpenFlow to improve backbone performance as well as to reduce backbone complexity and costs [Hö12].

In SDN-based networks, a NOS plays a major role. It is responsible for programming network devices which can be achieved pro-actively as well as re-actively in case of OpenFlow. Proactive programming means that before a switch receives a specific type of network packets, the NOS (possibly triggered by a SDN application) adds one or more flow rules to this switch. Such flow rules tell the switch in advance how to forward matching network packets. In contrast, re-active programming works as illustrated in Fig. 2. This figure shows a typical SDN setup including an OpenFlow switch, which is controlled by a NOS and connects two hosts h_1 and h_2. In case host h_1 sends a packet towards host h_2

while the switch is missing a corresponding forwarding decision (*e. g.*, a flow rule telling the swich to forward the packet to host h_2), this packet (or only its headers) is delegated to the NOS via a so called *packet-in message*. In a next step, the NOS processes this packet typically by passing it to the set of SDN applications which have registered for such events. Depending on the NOS's implementation, each of these SDN applications inspects this packet-in message, for example, in a sequential order, aiming to determine an adequate forwarding decision. Finally, the NOS informs the switch of this decision via a so called *packet-out message* and, according to this decision, the switch forwards the packet to host h_2. Thus, this illustrated network programming highlights the central role of a NOS. From a security perspective, this essential SDN component is also an interesting place for SDN malware, for example, in the form of a malicious SDN application.

In this paper, we give an overview of existing SDN malware, describe already proposed countermeasures and reveal problems which can arise when using those. Furthermore, we propose improvements aiming at filling the resulting security gaps. In particular, we focus on two security mechanisms, namely, sandboxing and policy checking. Current sandbox systems restrict SDN applications to access only a certain set of critical operations while access to all other (non-critical) operations is not controlled. Thereby, SDN applications can be prohibited to perform operations which can harm the NOS, for example, by crashing the NOS or manipulating internal data structures. We also discuss policy checkers which can intercept network programming attempts in order to check if the consequent network state would violate a security policy. This mechanism allows to drop harmful programming attempts and, thereby, keeps the network policy-conform. Since both approaches have issues which can lead to security breaches, we present security mechanisms to improve protection of a NOS against malicious SDN applications. In particular, we discuss how SDN applications can be isolated from the NOS in a stronger way by means of virtualization techniques. We also propose a mechanism which allows for robust policy checking, meaning that it runs completely independent from a NOS while minimizing hardware costs.

2 Background

Before we go into details, we outline the existing SDN malware to give an insight about the current state of malicious SDN applications and their capabilities. Furthermore, we briefly describe the basic functioning of security mechanisms which are able to counteract the SDN applications with malicious logic (*i. e.*, sandboxing and policy checking).

2.1 Malicious SDN Applications

To the best of our knowledge, no SDN malware has been found in the wild yet. However, researchers have already demonstrated what kind of attacks can be performed by malicious SDN applications [Po12, Po15, Sh14, RH15a, RH, RH15b].

In [Po12, Po15], a new technique is presented allowing attackers to evade flow rules which are existent in an OpenFlow switch. By adding specially crafted malicious flow rules, the authors demonstrate that an attacker can reach a network host although an existing drop rule explicitly denies such a connection. This technique is called *dynamic flow rule tunneling* and is based on standard OpenFlow instructions, *i. e.*, *set* and *goto*.

Other malicious SDN applications [Sh14, RH15a, RH] misuse critical operations on the system level in order to harm a NOS. For example, malicious SDN applications can easily crash SDN controllers, modify internal data structures or establish remote channels to retrieve shell commands from a C&C server, which in turn are executed on behalf of the NOS. In another example a malicious SDN application downloads and executes an arbitrary file with root privileges.

Recently, a SDN rootkit has been published [RH15b] for OpenDaylight which serves as a basis for multiple enterprise solutions [SDc]. This SDN rootkit heavily manipulates internal data structures and, thereby, gains control of the parts which are responsible for programming the network and reading the network's state. As a result, the authors demonstrate that an attacker is able to add and hide malicious flow rules as well as to remove legitimate flow rules without showing it to the administrator. Furthermore, an OpenFlow-based mechanism is presented which enables attackers to remotely communicate with the rootkit component running inside the NOS. This is interesting since a communication between hosts running on the data plane and the control plane is not a part of the SDN architecture.

2.2 Sandbox Systems

Currently, two sandbox systems for SDN controllers have been proposed. The first one runs SDN applications in separate processes and controls access to system calls [Sh14]. The other system utilizes Java security features to confine SDN applications inside of Java sandboxes [RH15a, RH, SDa]. The basic protection mechanism is illustrated in Fig. 3 and remains the same for both proposals.

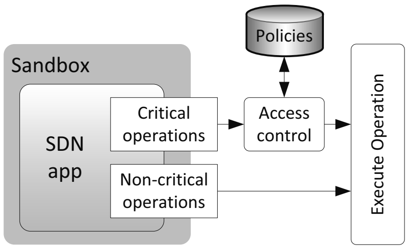


Fig. 3: Sandboxing

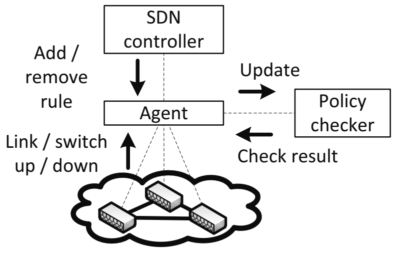


Fig. 4: Policy Checking

Each SDN application runs in a separate sandbox and access to critical operations (*i. e.*, system calls or sensitive Java operations) is controlled by a NOS. The sandbox configuration, *i. e.*, which SDN application is allowed to perform which critical operations, must be provided by a network administrator. In case he/she grants access to a critical operation, the corresponding SDN application can perform this operation while access is denied otherwise.

2.3 Policy Checking

Several studies introduce policy checking mechanisms for SDN-based networks [Ma11, KVM12, Kh12, Ka13]. The basic idea is to check whether the network state is conform to a given set of policies, for example, by looking for black holes, loops or security violations. While prior proposals struggle with performance issues, newer ones [Kh12, Ka13] provide the ability to perform policy checking at real time. This allows to intercept network programming attempts in order to check for policy violations first. Only in case of absent violations, corresponding flows rules are added to or removed from a network device. Fig. 4 illustrates this behavior for a policy checker called *NetPlumber* [Ka13] which is tested in Google's SDN-based WAN. In particular, a so called *NetPlumber agent* sits between the NOS and the programmable switches. This agent recognizes network state updates and in turn updates NetPlumber's internal model of the network. Such updates are triggered when a NOS adds or removes a flow rule and when a link or switch gets up or down. On each of these events, policy checks are initiated aiming at alerting an administrator in case a given policy is violated. Note that such an alert can occur before a policy violation takes place (*i. e.*, when adding or removing a flow rule) or soon after it (*i. e.*, when a link/switch gets up/down).

3 Problems of Current Protection Systems

3.1 Sandbox Systems

Concerning sandbox systems, malicious SDN applications can use potentially harmful operations only if access to them is granted explicitly. Current systems [Sh14, RH15a], however, support only low-level critical operations, *i. e.*, system calls and sensitive Java operations. This means that an administrator must have special knowledge in system security especially regarding operating system functioning and the Java programming language in order to estimate security-related consequences of granting access to a critical operation. Thus, the use of low-level based sandbox systems may lead to security breaches in practice as network administrators are rather experts in network than in system security and 50% to 80% of network outages are caused by misconfigurations [Ju08].

A recently proposed sandbox system [SDa] includes not only low-level but also high-level critical operations such as *reading the network topology* or *receiving a packet-in message*. Such operations are rather easy to understand by network administrators and corresponding security issues can be estimated more easily. However, supporting high-level critical operations does not mean that access to low-level critical operations can be denied by default. For example, SDN applications may depend on accessing the file system (*e. g.*, writing temporary files or reading configuration files), establishing network connections (*e. g.*, retrieving information stored on another network host) or using runtime capabilities such as creating processes or threads (*e. g.*, for concurrent programming or parallel execution). Denying access to such low-level critical operations by default would heavily limit SDN application developers in the way of implementing network functionality – which is the main purpose of a SDN application. In the best case, a sandbox system can deny access to a pre-configured set of low-level critical operations while allowing the administrator to decide on the remaining critical operations separately. However, this also limits SDN application developers to a level which may not be feasible in practice.

Even if we assume that network administrators are experts in both network and system security, it is difficult to determine the way (*i. e.*, benign or malicious) a critical operation is used in. But access to a critical operation can only be denied by default if it is exclusively used in a malicious way. To illustrate the problem, let us have a look at Java reflection. It can be used to read or manipulate internal data structures of a Java-based NOS. Because of this, one might guess that access to corresponding critical operations should be denied by default. In fact, Java reflection has already been used maliciously to implement a SDN rootkit [RH15b]. However, legitimate SDN applications may also use Java reflection but in a benign way. In particular, we have analyzed several SDN kernel applications provided by the HP app store. At the time of writing, we find that 6 out of 11 examined SDN applications also use Java reflection. At a first glance, we could not find a single Java reflection related critical operation which is used by the aforementioned SDN rootkit but never by the benign SDN applications.

Consequently, configuration of current sandbox systems with malicious logic prevention is rather difficult. Taking this into account, network administrators may require more effective systems in order to improve protection of a NOS against malicious SDN applications.

3.2 Policy Checking

With regard to policy checking, adverse network manipulations (*e. g.*, a malicious flow rule) can be prohibited by checking at real time if a network event (*e. g.*, adding a new flow rule) violates one or more given security policies. Current real time policy checkers run either inside [Kh12] or outside a NOS [Ka13]. Since malicious SDN applications (such as a SDN rootkit) can compromise a NOS including its security checks [RH15b], we focus on policy checking performed outside the NOS.

For example, NetPlumber [Ka13] is implemented as a network service which is externally triggered by a NetPlumber agent to perform policy checks. Where this agent should be implemented is not discussed in detail by the authors. In fact, they simply use local files instead of network connections in order to load network updates into the NetPlumber system during evaluation. In general, there are several options where such an agent can be implemented: (i) inside a NOS, (ii) outside a NOS but running on the same host and (iii) outside a NOS and running on a separate host. In the first case only a few additional changes within a NOS are necessary for triggering policy checking before a flow rule is sent to a switch. Obviously, this can be achieved easily and cost-effective. The second option is more complicated but still does not require additional hardware as a separate proxy process or network packets interception on kernel level can achieve network programming interception. Most complex and expensive is the last option: running the agent on a separate hardware. For example, a backup component for a NOS is required to handle breakdowns or maintenance-related shutdowns. Another cost-driving factor is the network interface connecting SDN switches and a NOS, which must handle network events at high speed [Ka09]. Furthermore, in case of long distance networks, SDN controllers must run at different locations to avoid latency issues [HSM12]. Additional hardware for such scenarios and extra complexity in terms of additional operational costs can cause additional expenses.

As a major goal of SDN is cost degression, such additional expenses may not be an option for network operators. However, running policy checks independently from the NOS is highly desirable for security reasons.

4 Potential Countermeasures

In order to fill the security gaps caused by the aforementioned problems, we propose the following countermeasures.

4.1 Strong Isolation of SDN Applications

Considering the issues described in Section 3.1, we need a mechanism which counteracts malicious SDN applications while assuming that sandboxing is insufficient in practice. In order to compensate that, we propose the following NOS architecture which is based on isolation of SDN applications by means of virtualization techniques. This is new with respect to SDN controllers and it enables a more stringent way of separating SDN applications compared to existing systems.

Our architecture is illustrated in Fig. 5 and includes several protection domains each running separately, for example, in a virtual machine. Inter-domain communication is only allowed between the SDN controller domain and each application domain while this communication is restricted to high-level and SDN-related operations. Assuming that virtualization presumes domain isolation, the big benefit of this architecture is that critical low-level operations are executed within each protection domain, thus, negative consequences are limited to the affected domain only.

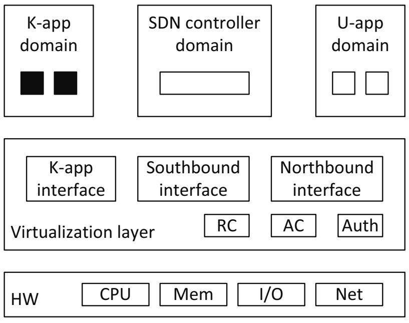


Fig. 5: Strong Isolation of SDN Apps

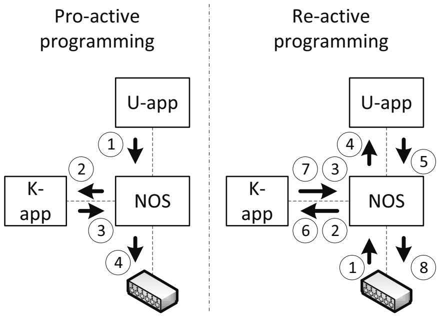


Fig. 6: Critical Data Path

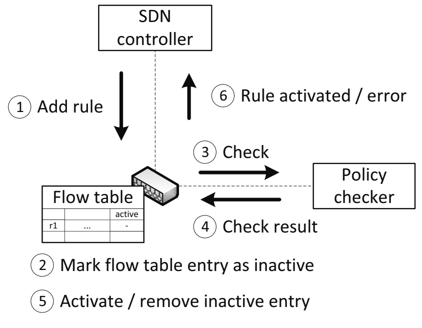
In particular, we run one or more SDN applications in a separate application domain instead of starting them in separate processes. Thereby, SDN user applications (U-apps) communicate via the northbound interface with the SDN controller, SDN kernel applications (K-apps) use a NOS-specific interface (K-app interface) for communication with the NOS, and the SDN controller uses the southbound interface to interact with the SDN-based network. In addition, a virtualization layer provides Authentication (Auth), Resource Control (RC) and Access Control (AC) mechanisms allowing to authenticate components, assign hardware resources to each domain and control access to operations related to the aforementioned interfaces. In case of the malicious use of critical operations, this architecture limits negative consequences in a way not available in current systems. For example, if a SDN rootkit uses Java reflection and the corresponding sandbox allows such critical operations, today's Java-based SDN controllers can be compromised despite the use of sandboxes. Given our architecture, this is not possible while SDN application developers remain unrestricted in inventing new applications.

The main challenge in this respect is to implement network programming efficiently as this is the main purpose of a NOS. Therefore, Fig. 6 shows the critical data paths for pro-active and re-active network programming, separately. In case of pro-active programming, a flow rule generated by a SDN user application must be transferred from an application-domain to the controller-domain (①). Depending on the SDN controller, a SDN kernel application may be used to receive this flow rule (②), encapsulate it in a southbound protocol dependent format and send it back to the NOS (③). Then, the NOS must transmit this packet to the affected switch(es) (④). In case of re-active programming, the critical data path consists of some additional steps. When a network packet is delegated to a NOS (①), the NOS must pass it to the component which implements the southbound protocol, for example, a SDN kernel application (②). After extracting the needed information, it must be passed to the SDN user applications which are registered for such tasks (③ and ④). Next, after determining an adequate forwarding decision it must be sent to the affected switches which works similar to pro-active programming (⑤, ⑥, ⑦ and ⑧). As performance of network programming is of high importance, the biggest challenge is to reduce the overhead in order to achieve practicable results.

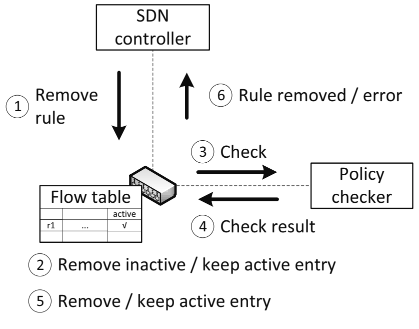
4.2 Robust Policy Checking

Checking the network state against policy violations at real time is challenging especially in case of a compromised SDN controller. Besides running policy checker agents on separate hardware as described in Section 3.2, we propose an alternative aiming at saving the hardware costs. As illustrated in Fig. 7(a) and 7(b), we introduce a policy checking mechanism which is completely independent from a NOS, thus, robust against a compromised NOS. The basic idea is to treat a new flow table entry as inactive until a policy checking instance signals that no policy is violated.

Fig. 7(a) illustrates the work flow of adding a flow rule. In this case, a SDN controller programs network devices as usual, for example, via OpenFlow (①). However, the corresponding programming command not only adds a flow table entry but also marks it as



(a) Add Rule



(b) Remove Rule

Fig. 7: Robust Policy Checking

inactive (②). In the next step, a switch acts as a policy checker agent and triggers a policy checker to validate the absence of policy violations when adding a new flow rule (③). Depending on the check result (④), the switch either activates the flow table entry or removes it from the flow table (⑤). In the final step, the switch responds either by an error message or by a *flow activated* message telling the NOS that the programmed flow rule is policy-conform (⑥). In case of removing a flow rule, the work flow remains similar as shown in Fig. 7(b). Receiving a command to remove a flow rule (①), the switch checks if the corresponding flow table entry is active or not (②). In case of inactive entry, the switch removes it and returns a *flow removed* message to the NOS (⑥). In the opposite case, the flow table entry remains active until policy checker signals no policy violations by removing the flow rule (③ and ④). After this confirmation, the active entry is removed (⑤) and a flow removed message is sent to the NOS (⑥). In case of a policy violation, the corresponding flow entry remains active and a corresponding error message is sent to the NOS (⑥).

In contrast to current real time policy checkers, the mechanism does not need extra hardware in order to run completely independent of a NOS. In case of OpenFlow, however, this mechanism requires several changes: (i) OpenFlow must support roles not only for SDN controllers (*e. g.*, master and slave) but also for policy checkers; (ii) flow table entries must be extended by a field indicating active vs. inactive entries; (iii) processing of network packets must be adapted such that only active flow table entries are considered when finding a matching flow table entry; and (iv) OpenFlow switches must interact with a policy checker in order to process network programming. For reliability reasons, a switch should be able to interact with multiple policy checkers to avoid single points of failures.

5 Related Work

The problem of malicious SDN applications was first addressed by Porras *et al.* [Po12]. The authors present a new technique called *dynamic flow rule tunneling* which enables attackers to bypass existent flow rules by exploiting standard OpenFlow instructions. Later on, Shin *et al.* [Sh14] and Röpke *et al.* [RH15a] present SDN applications with rudimentary malicious logic. For this purpose, the authors exploit the fact that many SDN controllers allow a SDN application to execute any kind of critical operations without limiting access to them. Recently, Röpke *et al.* [RH15b] demonstrate how a sophisticated SDN malware can compromise SDN controllers. In particular, the authors provide a SDN rootkit for the popular OpenDaylight controller which serves as a foundation for several enterprise solutions.

Despite the fact that no malicious SDN application has been found in the wild yet, we take the findings of previous studies as examples of upcoming attack scenarios. This motivated us to examine current protection mechanisms regarding their efficiency against sophisticated malicious SDN applications. Sandboxing as well as policy checking have been already discussed by Röpke *et al.* [RH15b], but we discuss the potential problems in detail and present valuable solutions.

Our first proposal is mainly motivated by security systems such as Qubes [RW10] and Bromium [Br] which use virtualization techniques to isolate applications from common operating systems like Linux and Windows. Note that using such a technique does not provide isolation properties by default as it was shown recently [XE]. However, vulnerabilities which allow attackers to break out of a VM are rather rare while escaping from a process environment is commonly used by malware, for example, to infect other processes. Our second proposal is motivated by the fact that a compromised SDN controller can manipulate policy checking [RH15b]. We therefore extend OpenFlow [Opa] in order to trigger policy checks independently from a SDN controller. One may think that adding functionality to a network device is against the paradigm of SDN which aims at decoupling control functions from such elements. However, the OpenFlow specification already includes functions like checking for overlapping flow rules. Moreover, adding security functions on the data plane has been implemented efficiently before [Sh13b].

6 Conclusions

In this paper, we discuss problems of state-of-the-art security mechanisms which help in protecting SDN controllers against malicious SDN applications. We argue that current sandbox systems and policy checking mechanisms indicate problems when considering sophisticated SDN malware such as a SDN rootkit. In order to fill the corresponding security gaps, we propose two improvements. On the one hand, we introduce a SDN controller design based on virtualization techniques which enables a strong isolation of SDN applications from a SDN controller. On the other hand, we present an enhanced way of policy checking which runs independently from the SDN controller and remains robust even if a NOS gets compromised. Each of our proposals improves the security of SDN-based networks especially against malicious SDN applications. This is particularly important as we believe that SDN malware will appear soon considering the fast development of SDN.

References

[Al15] Ali, Syed Taha; Sivaraman, Vijay; Radford, Adam; Jha, Sanjay: A Survey of Securing Networks using Software Defined Networking. IEEE Transactions on Reliability, 2015.

[Br] Bromium: , Bromium. www.bromium.com.

[Fl] Floodlight project: , Floodlight. floodlight.openflowhub.org.

[Ga14a] Gartner: , Gartner Identifies the Top 10 Strategic Technology Trends for 2015. http://www.gartner.com/newsroom/id/2867917, 2014.

[Ga14b] Gartner: , Predicting SDN Adoption. http://blogs.gartner.com/andrew-lerner/2014/12/08/predicting-sdn-adoption/, 2014.

[Gu08] Gude, Natasha; Koponen, Teemu; Pettit, Justin; Pfaff, Ben; Casado, Martín; McKeown, Nick; Shenker, Scott: NOX: Towards an Operating System for Networks. ACM SIGCOMM Computer Communication Review, 2008.

[HP13] Hewlett-Packard: , HP Open Ecosystem Breaks Down Barriers to Software-Defined Networking. www8.hp.com/us/en/hp-news/press-release.html?id=1495044#.Vh-AbR_HnCI, 2013.

[HSM12] Heller, Brandon; Sherwood, Rob; McKeown, Nick: The controller Placement Problem. In: ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking. 2012.

[Hö12] Hölzle, Urs: , OpenFlow @ Google. Open Networking Summit, 2012.

[Ju08] Juniper Networks: , What's behind network downtime? www-935.ibm.com/services/au/gts/pdf/200249.pdf, 2008.

[Ka09] Kandula, Srikanth; Sengupta, Sudipta; Greenberg, Albert; Patel, Parveen; Chaiken, Ronnie: The nature of data center traffic: measurements & analysis. In: Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference. 2009.

[Ka13] Kazemian, Peyman; Chang, Michael; Zeng, Hongyi; Varghese, George; McKeown, Nick; Whyte, Scott: Real Time Network Policy Checking Using Header Space Analysis. In: USENIX Symposium on Networked Systems Design and Implementation. 2013.

[Kh12] Khurshid, Ahmed; Zhou, Wenxuan; Caesar, Matthew; Godfrey, P: Veriflow: Verifying Network-Wide Invariants in Real Time. ACM SIGCOMM Computer Communication Review, 2012.

[KVM12] Kazemian, Peyman; Varghese, George; McKeown, Nick: Header Space Analysis: Static Checking for Networks. In: USENIX Symposium on Networked Systems Design and Implementation. 2012.

[Ma11] Mai, Haohui; Khurshid, Ahmed; Agarwal, Rachit; Caesar, Matthew; Godfrey, P; King, Samuel Talmadge: Debugging the Data Plane with Anteater. ACM SIGCOMM Computer Communication Review, 2011.

[Mc08] McKeown, Nick; Anderson, Tom; Balakrishnan, Hari; Parulkar, Guru; Peterson, Larry; Rexford, Jennifer; Shenker, Scott; Turner, Jonathan: OpenFlow: Enabling Innovation in Campus Networks. ACM SIGCOMM Computer Communication Review, 2008.

[ON] ONOS Project: , Open Network Operating System. onosproject.org.

[Opa] Open Networking Foundation: , OpenFlow Switch Specification, Version 1.4.0. www.opennetworking.org.

[Opb] Open Networking Foundation: , SDN/OpenFlow Products. `www.opennetworking.org/sdn-openflow-products`.

[Opc] OpenDaylight Project: , OpenDaylight. `www.opendaylight.org`.

[Po12] Porras, Philip; Shin, Seungwon; Yegneswaran, Vinod; Fong, Martin; Tyson, Mabry; Gu, Guofei: A Security Enforcement Kernel for OpenFlow Networks. In: ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking. 2012.

[Po15] Porras, Phillip; Cheung, Steven; Fong, Martin; Skinner, Keith; Yegneswaran, Vinod: Securing the Software-Defined Network Control Layer. In: Symposium on Network and Distributed System Security. 2015.

[RH] Röpke, Christian; Holz, Thorsten: On Network Operating System Security. To appear in International Journal of Network Management.

[RH15a] Röpke, Christian; Holz, Thorsten: Retaining Control Over SDN Network Services. In: International Conference on Networked Systems (NetSys). 2015.

[RH15b] Röpke, Christian; Holz, Thorsten: SDN Rootkits: Subverting Network Operating Systems of Software-Defined Networks. In: Symposium on Recent Advances in Intrusion Detection. 2015.

[RW10] Rutkowska, Joanna; Wojtczuk, Rafal: Qubes OS architecture. Invisible Things Lab Tech Rep, 2010.

[SDa] SDN security project: , Secure-Mode ONOS. `sdnsecurity.org`.

[SDb] SDxCentral: , Comprehensive List of SDN Controller Vendors & SDN Controllers. `www.sdxcentral.com/resources/sdn/sdn-controllers/sdn-controllers-comprehensive-list/`.

[SDc] SDxCentral: , SDN Controllers Report. `www.sdxcentral.com/reports/sdn-controllers-report-2015/`.

[Sh13a] Shin, Seungwon; Porras, Phillip; Yegneswaran, Vinod; Fong, Martin; Gu, Guofei; Tyson, Mabry: FRESCO: Modular Composable Security Services for Software-Defined Networks. In: Symposium on Network and Distributed System Security. 2013.

[Sh13b] Shin, Seungwon; Yegneswaran, Vinod; Porras, Phillip; Gu, Guofei: AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-Defined Networks. In: ACM Conference on Computer and Communications Security. 2013.

[Sh14] Shin, Seungwon; Song, Yongjoo; Lee, Taekyung; Lee, Sangho; Chung, Jaewoong; Porras, Phillip; Yegneswaran, Vinod; Noh, Jiseong; Kang, Brent Byunghoon: Rosemary: A Robust, Secure, and High-Performance Network Operating System. In: ACM Conference on Computer and Communications Security. 2014.

[XE] XEN project: , Security Advisory XSA-148. `xenbits.xen.org/xsa/advisory-148.html`.