Covert Channels in SDN: Leaking Out Information from Controllers to End Hosts

Jiahao Cao[1,3], Kun Sun[3], Qi Li[2], Mingwei Xu[1,2(✉)], Zijie Yang[1], Kyung Joon Kwak[4], and Jason Li[4]

[1] Department of Computer Science and Technology, Tsinghua University, Beijing, China
xmw@cernet.edu.cn
[2] Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China
[3] Department of Information Sciences and Technology, George Mason University, Fairfax, USA
[4] Intelligent Automation Inc., Rockville, USA

Abstract. Software-Defined Networking (SDN) enables diversified network functionalities with plentiful applications deployed on a logically-centralized controller. In order to work properly, applications are naturally provided with much information on SDN. However, this paper shows that malicious applications can exploit basic SDN mechanisms to build covert channels to stealthily leak out valuable information to end hosts, which can bypass network security policies and break physical network isolation. We design two types of covert channels with basic SDN mechanisms. The first type is a high-rate covert channel that exploits SDN proxy mechanisms to transmit covert messages to colluding hosts inside SDN. The second type is a low-rate covert channel that exploits SDN rule expiry mechanisms to transmit covert messages from SDN applications to any host on the Internet. We develop the prototypes of both covert channels in a real SDN testbed consisting of commercial hardware switches and an open source controller. Evaluations show that the covert channels successfully leak out a TLS private key from the controller to a host inside SDN at a rate of 200 bps with 0% bit error rate, or to a remote host on the Internet at a rate of 0.5 bps with less than 3% bit error rate. In addition, we discuss possible countermeasures to mitigate the covert channel attacks.

Keywords: SDN · Covert channels · Information leakage

1 Introduction

Software-Defined Networking (SDN) enables network innovations by decoupling control and data planes with a logically-centralized controller managing the entire network. As an emerging network paradigm, SDN is being widely adopted

S. Chen et al. (Eds.): SecureComm 2019, LNICST 304, pp. 429–449, 2019.
https://doi.org/10.1007/978-3-030-37228-6_21

in enterprise data centers, cloud networks, and virtualized environments [29]. Driving the popularity of SDN is that various applications can be developed and installed on the controller to enrich diversified network functionalities, such as load balancing [14], traffic engineering [19], and network security forensics [48]. SDN with plentiful applications greatly meets the need of industry to build dynamic, agile, and programmable networks.

In order to run applications properly, the controller directly provides many types of network information to them. For example, a routing application gets detailed network topology and positions of hosts from the controller to schedule routing paths. Moreover, an application can get more information than what it needs by actively retrieving data objects [47,50] on controllers. For example, an application can know security policies of the network by querying flow rules with the `FlowRuleService` object. Consequently, malicious SDN applications [23,32, 49,51] running on controllers can also possess many types of SDN information either by the direct offer of controllers or actively retrieving information. They may transmit the information to remote attackers for launching efficient and stealthy attacks. For instance, with the knowledge of detailed security policies in the network, attackers' hosts can craft special attacking traffic to accurately bypass network security policies.

To prevent potential information leakage from the malicious applications to colluding hosts, SDN environments usually enforce strict access control policies and deploy security devices to enhance the network security [1,2,33]. For example, unauthorized connections between an SDN application and a remote host can be cut off by firewalls, and abnormal TCP or UDP connections can be detected with a modern Network Intrusion Detection System (NIDS). Moreover, SDN allows building a separated physical network for transmitting control traffic between controllers and switches, i.e., out-of-band control [16,50]. Therefore, the controller is well isolated from end hosts that attempt to communicate with the malicious applications running on the controller. Existing mechanisms make it difficult for a malicious application to directly transmit valuable data to colluding end hosts.

In this paper, we study how a malicious application on SDN controllers can transmit SDN information to an end host with covert channels, which can bypass existing security policies and succeed even if the controller is isolated in a separated physical network. Our key insight is that SDN switches forward data traffic among end hosts and communicate with controllers to enforce policies at the same time. Thus, the switches may be the potential communication bridges between an end host and an application running on controllers. We design two types of covert channels with basic SDN mechanisms. The first type is high-rate covert channels between an application and a host inside SDN, which exploits the SDN proxy mechanism to encode covert information into response packets. The second type exploits the SDN rule expiry mechanism to construct low-rate covert channels between an application and any remote host on the Internet. Covert information is encoded into delays of data packets by delaying reinstallation of flow rules. Although previous studies [41,44,49,52] provide defense to prevent malicious applications from disrupting network operations, such as application

isolation [44] and permission control [41,49,52], they fall short with respect to the prevention of covert channels built by basic SDN mechanisms.

We also perform a comprehensive study on what information an application can obtain on five major SDN controllers, i.e., `OpenDaylight`, `ONOS`, `Floodlight`, `RYU`, and `POX`. We first summarize three viable collection channels for an unprivileged application to collect information on the controller, namely, possessing information by design in order to work properly, exploiting system misconfiguration, and actively reading shared data objects. Then we show a list of SDN information that can be collected by an application, such as controllers' TLS keys and certifications, network security policies, routing policies, inter-host communication patterns, and network topology information.

We develop the prototypes of both covert channels in a real SDN testbed consisting of commercial hardware switches and an open source controller. Experimental results show that the covert channels built by the SDN proxy mechanism can successfully transmit a TLS private key from the controller to a host inside SDN at a rate of 200 bps with 0% bit error rate. In addition, we rent six hosts located in different positions to build the covert channels between an SDN application and the hosts on the Internet. Our experiments show that by exploiting the SDN rule expiry mechanism, a TLS private key can be transmitted from our SDN controller to the remote hosts at a maximum rate of 0.5 bps with less than 3% bit error rate. With the low bit error rate, the entire TLS private key can be fully recovered by adding lightweight error correction codes.

To summarize, our paper makes the following contributions:

- We develop two new types of SDN covert channels that use the basic SDN proxy and rule expiry mechanisms to transmit information from the controller to hosts inside SDN or outside SDN, respectively.
- We perform a comprehensive study on what information an SDN application may collect on five major SDN controllers.
- We conduct experiments in a real SDN testbed to demonstrate the feasibility and effectiveness of the identified covert channels.

2 Threat Model

We assume a malicious application has been installed on an SDN controller. Previous studies [23,32,49,51] have demonstrated that malicious applications can be installed in many ways, such as phishing with repackaging or redistributing applications [51], exploiting particular vulnerabilities of controllers [23], and fooling network administrators into downloading malicious applications from SDN App Store [49]. We do not assume the malicious application can compromise the controller, switches, and control channels protected by TLS [11]. The malicious application may correctly perform its designated network functionalities, e.g., generating correct flow rules as a routing application; however, it may also leverage basic SDN operations to build covert channels with a colluding host at the same time. Moreover, we consider two scenarios for the locations of colluding hosts. First, if the attacker may rent a host in the target SDN, e.g., renting a

virtual host in SDN-based cloud networks [6], it can control the host inside SDN as the colluding host. Second, if the target SDN includes a server that provides services for public access, e.g., websites, the attacker may use any host on the Internet as the colluding host.

We assume SDN can use either in-band control [16,50], where the control traffic between controllers and switches is delivered with data traffic over the same network infrastructure, or out-of-band control [16,50], where a separated physical network is dedicated for delivering control traffic. The out-of-band control can prevent an end host from accessing controllers since they are isolated in different physical networks. In addition, SDN may also enforce strict access control policies to protect controllers from unauthorized communications. For example, SDN firewalls can block the TCP connections between an SDN application and a host. Therefore, the attackers are motivated to build reliable and stealthy communication channels to leak valuable data from SDN applications to a colluding host.

3 Covert Channel Attacks

The attack goal is to leak out information collected by an application to a remote host. The attack should bypass existing defense mechanisms such as firewalls, intrusion detection system, and access control policies. Moreover, it should work in SDN with out-of-band control that isolates controllers in a separated physical network which provides no communication between controllers and end hosts.

We develop two types of covert channels with basic SDN operations for two scenarios, which do not trigger unusual control messages or flood packets. The first type exploits SDN proxy mechanisms [4,7–10] to achieve a high-rate covert channel. It transmits covert messages to colluding hosts inside SDN. When no colluding host is possessed or compromised in SDN, the second type leverages the SDN rule expiry mechanisms [11] to build a low-rate covert channel with a colluding host on the Internet. It only requires the colluding host to have access to a public service in the target SDN, e.g., visiting a public website.

3.1 Covert Channels with SDN Proxy Mechanisms

As SDN separates control and data planes, the SDN data plane devices (e.g., SDN switches) themselves usually cannot implement proxy functions like traditional routers. Thus, applications running on controllers leverage SDN proxy mechanisms to provide network proxy functionalities for end hosts, such as `ARP Proxy` [4,8,9], `NDP Proxy` [7], and `DHCP Proxy` [3]. Although applications enable many types of proxy functionalities, they perform similar behaviors with basic `packet-in` and `packet-out` operations to implement the functionalities. For simplicity but without loss of generality, we use `ARP Proxy` as an example to illustrate SDN proxy mechanisms.

As shown in Fig. 1, when an ARP request packet sent by a host arrives at an ingress switch, it is encapsulated into a `packet-in` message and sent to the

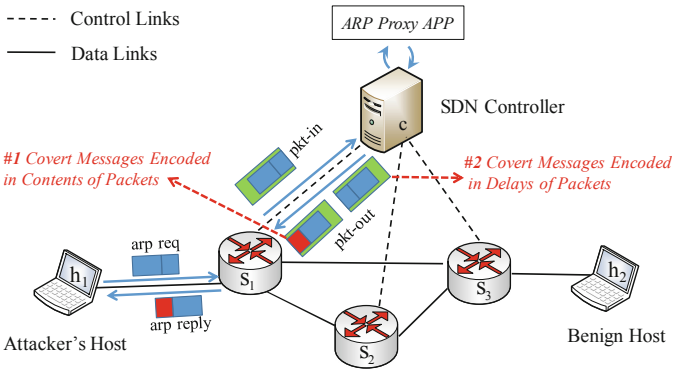


Fig. 1. An example of covert channels built by SDN proxy mechanisms.

controller. The controller extracts the request packet and dispatches it to applications that register `packet-in` event listener. The application enabling the ARP proxy finds an entry where the IP address matches the target IP address in the ARP request packet. It then creates an ARP reply packet that is encapsulated into a `packet-out` message in order to be sent back to the ingress switch. The ingress switch decapsulates the packet and generates an ARP response packet to the original host. Other applications enabling different proxy functionalities perform similar behaviors like `ARP Proxy` with different request and reply packets encapsulated in `packet-in` and `packet-out` messages, respectively. For example, an NDP neighbor solicitation packet and an NDP neighbor advertisements are encapsulated in `packet-in` and `packet-out` messages for `NDP Proxy`, respectively.

A malicious application can exploit the SDN proxy mechanism to build covert channels with a host inside the target SDN network to leak out information, though the application and the host cannot directly communicate with each other due to security policies or network isolation. There are two techniques that can be applied to build covert storage and timing channels, respectively.

Packet Encoding. As the response packet is generated by the application enabling proxy functionalities, additional information can be encoded into the packet. Thus, it allows building a covert storage channel between the application and a colluding host. For example, an attacker's host h_1 can send an ARP request packet to pretend to query the MAC address of some host in Fig. 1. However, it actually aims to stealthily receive SDN information from the application. Once the application receives the ARP request packet, it generates an ARP response packet with extra covert information encoding into some bits in the packet, which is shown in the red parts of `packet-out` messages in Fig. 1. Here, the bits can be reserved bits or unused bits in packets. For example, the last 144 bits of padding

in an ARP packet[1] or the reserved 29 bit in an NDP neighbor advertisements packet [39]. Once the attacker's host h_1 receives the packet, it can extract the information by inspecting the content of the packet. Moreover, another way is to encode the information into some existing header fields of the ARP response packet, such as Target IP Address and Target MAC Address. However, in order to maintain the normal functionalities of the application and incur no anomalies for other hosts in this case, the application should only replace the original values of Target IP and MAC Address in ARP response packets for the attacker's host. ARP response packets for other hosts should maintain original values.

Response Timing. Covert channels built by such technique are stealthier, as no obvious information is encoded in headers or payload of packets. The main idea is simple but effective. As the attacker's host can measure the round-trip time (RTT) between sending a request packet and receiving a corresponding response packet, the application can deliberately delay responding a packet with different time to signal a "1" or a "0". Thus, a covert timing channel is built between the application and a colluding host. For example, suppose the host h_1 in Fig. 1 receives three response packets with 10 ms, 5 ms, and 6 ms RTTs, respectively, 3 bits covert messages "100" can be interpreted from the RTTs. In practice, the threshold of RTTs on distinguishing between "1" and "0" can significantly affect the accuracy of interpreting covert messages. We will show how to optimally encode and decode covert timing messages in Sect. 3.3.

The rate of above covert channels mainly depends on the number of request packets that a colluding host can send per second. Thus, an end host can adjust the rate at will to make a trade-off between the rate and the stealthiness. For example, the host can choose to send a request packet every 10 s. Although the transmission rate for the covert timing channel is 0.1 bps in the case, the behavior of delivering covert messages is hard to be detected since the rate is low and there are few packets transmitting covert messages per second.

3.2 Covert Channels with SDN Rule Expiry Mechanisms

In SDN, each flow is forwarded according to flow rules. However, due to the limited storage space of flow rules in switches [27], rule expiry mechanisms are introduced to efficiently use the storage space. The mechanisms associate each flow rule with two types of timeouts, i.e., hard timeout and idle timeout, indicating the expiration time of the rule. A flow rule will be removed by switches either after the given number of seconds of hard timeout no matter how many packets it has matched or when it has matched no packets within the given number of seconds of idle timeout. Therefore, when packets of flows arrive at switches where related flow rules have expired, the packets are buffered in switches and wait for switches to query controllers to reinstall flow rules.

We discover that the basic rule expiry mechanism allows malicious applications to build covert channels with a remote host on the Internet. Figure 2 shows

[1] The padding aims to bloat the ARP frame to the 64-byte length which is the minimum required length of an Ethernet frame.

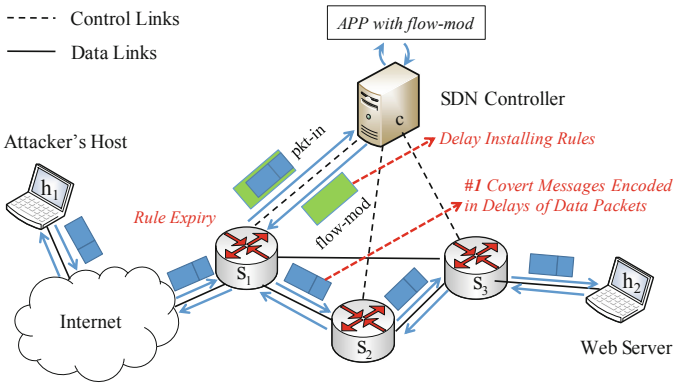


Fig. 2. An example of a covert channel built by SDN rule expiry mechanisms.

a typical case. The host h_2 is a web server which provides public services, such as websites. The host h_1 on the Internet visits the websites provided by h_2. Thus, a HTTP request flow is sent from h_1 to h_2 and a related HTTP reply flow is sent from h_2 to h_1 to load the web page. Once the web page is successfully loaded, the flow stops transmission. After some time, if h_1 visits another web page from the website, new HTTP flows between the two hosts are generated. However, if the interval between sending the new HTTP request flow and sending the previous HTTP request flow exceeds the value of rule timeouts, existing matched flow rules disappear and thus the HTTP request packets have to be buffered in the switch s_1. Meanwhile, the switch s_1 copies a packet from the flow and encapsulates it into a `packet-in` message to query rule installation from the SDN controller. The application on the controller then reinstalls the flow rules with a `flow-mod` message according to the analysis of the `packet-in` message. Finally, packets in the switch's buffers are forwarded according to the flow rules.

During the process, the application responsible for installing flow rules can deliberately delay some time before generating `flow-mod` messages for flow rule installation. The delays of rule installation will be reflected in the forwarding delays of the packets that are temporarily buffered in switches, which is shown in Fig. 2. Thus, the application can encode covert messages into the RTTs of HTTP packets by deliberately delaying the installation of flow rules with different values to signal a "1" or a "0". The colluding host h_1 on the Internet can extract covert information according to the RTT of the first packet of each new HTTP flow.

In order to ensure that the first packet of each new HTTP flow can be encoded covert information, the colluding host should wait for enough time to make rules expired before sending a new flow. According to the previous work [18], most timeout settings of rule expiration are usually less than 60 s and also can be inferred in advance by sending probe packets. In addition, the threshold of RTTs of HTTP packets on distinguishing between "1" and "0" will affect the accuracy of interpreting covert timing messages, we show how to optimally encode and decode covert timing messages in Sect. 3.3.

We do not transmit covert information with the packet encoding technique in the covert channel built by the SDN rule expiry mechanism. In general, the first packet that is sent to the SDN controller is only used to reinstall rules, which will not be sent back to the original switch since there is an identical packet in the switch's buffers. Hence, encoding bits in the packet fails to transmit information to remote hosts. Although the packet is sent back to the switch which fails to buffer the packet due to no buffer space available [11], it is a rare case in practice and thus cannot be exploited to build a stable and persistent covert channel.

3.3 Encoding and Decoding Covert Timing Messages

Both the covert timing channels built by the proxy and rule expiry mechanisms require a threshold of RTTs to decide which RTTs indicate "1" and which RTTs indicate "0". The threshold should be optimal to improve communication accuracy as much as possible. Moreover, the threshold should be adaptive. In other words, the threshold adaptively changes when applications transmit covert messages to different colluding hosts that have different network delays. For example, according to our measurements, the HTTP packets of a colluding host located in New York have an average RTT of 215 ms. However, the HTTP packets of another colluding host located in Beijing have an average RTT of 3 ms. The optimal thresholds of RTTs on distinguishing between "1" and "0" in the two colluding hosts are significantly different.

In order to adaptively and optimally decode timing information, we apply the Manchester code [5] to encode covert messages into the delays of packets. The Manchester code encodes 1-bit "1" into 2-bit "10" and 1-bit "0" into 2-bit "01". Thus, when an application signals "1" with the Manchester code, it encodes the information into a pair of packets by delaying the first packet. When signaling "0" with the Manchester code, it encodes the information into a pair of packets by delaying the second packet. Whenever a colluding host receives a pair of packets, "1" can be decoded if the RTT of the first packet is larger than the RTT of the second packet. Otherwise, "0" can be decoded from the RTTs of the two packets. Formally, consider an application encoding n bits in n pair of packets, the 1-bit information b_i from the i_{th} pair of packets can be decoded as follows:

$$\forall i \in \{1, 2, ..., n\},\ b_i = \begin{cases} \text{“1”},\ if\ \eta_i^1 > \eta_i^2 \\ \text{“0”},\ if\ \eta_i^1 \leq \eta_i^2 \end{cases} \tag{1}$$

Here, η_i^1 and η_i^2 denote the RTT of the first packet and the second packet in the i_{th} pair of packets, respectively.

3.4 Covert Channel Analysis

The Number of Exploitable SDN Applications. Applications building covert channels must have the ability to leverage SDN proxy or rule expiry mechanisms. Proxy applications and routing applications naturally can leverage SDN proxy mechanisms and rule expiry mechanisms, respectively. Actually, any

Table 1. The number of exploitable applications

Controller	Total APPs	Type I APPs	Type II APPs	Type I ∩ II APPs[b]
`OpenDaylight Neon`[a]	13	9	6	5
`ONOS v2.1.0-rc1`	97	26	23	22
`Floodlight v1.2`	29	13	12	10
`RYU v4.31`	28	16	19	16
`POX eel version`	18	9	11	9

[a]For OpenDaylight, we only count the applications with the openflowplugin implementation.
[b]The applications belong to Type I and Type II simultaneously.

applications (defined as **Type I APPs**) with permissions of listening `packet-in` messages and performing `packet-out` operations can leverage SDN proxy mechanisms. Any applications (defined as **Type II APPs**) with permissions of listening `packet-in` messages and performing `flow-mod` operations can leverage rule expiry mechanisms. We find that the permissions are easily met as they are basic requirements for running many applications. We investigate bundled applications on five popular SDN controllers with the latest versions. Table 1 shows the total bundled applications on controllers and the number of applications among them that may leverage the two mechanisms to build covert channels.

Transmission Rate. The covert channels above are built with two different SDN mechanisms. For simplicity, we use a_s and a_t to denote the covert channels built by the SDN proxy mechanism with packet encoding and response timing, respectively. We use b_t to denote the covert channel built by the SDN rule expiry mechanism. Totally, there are three covert channels. Different covert channels can transmit information at different rates. We give an analysis of their transmission rates. We use v_x, where $x \in \{a_s, a_t, b_t\}$, to denote the transmission rate of the covert channel x. For covert channels built with the SDN proxy mechanism, covert information can be transmitted once a request packet is sent. For covert channels built with the SDN rule expiry mechanism, only 1-bit covert information can be transmitted every timeout value after a flow rule is expired. Moreover, the transmission rate is reduced by half due to the Manchester code. Thus, we have:

$$\begin{cases} v_{a_s} = p \cdot l \\ v_{a_t} = \dfrac{p}{2} \\ v_{b_t} = \dfrac{1}{t \cdot 2} \end{cases} \tag{2}$$

Here, p denotes the number of request packets sent per second, l denotes the number of coded bits for transmitting covert information in a packet, t denotes the minimum value of hard timeout and idle timeout of a flow rule.

A host may send millions of request packets per second to make the application transmit covert information as fast as possible. However, most existing hardware switches can only generate thousands of `packet-in` messages per sec-

ond [34], which limits the maximum transmission rate. If we use C to denote the maximum rate of generating `packet-in` packets for a single switch and L to denote the maximum available bits that can be coded in a packet, we have: $v_{a_s}^{max} = C \cdot L$, $v_{a_t}^{max} = \frac{C}{2}$. Moreover, according to the OpenFlow specification [11], the minimum valid value for timeout is 1 s. Thus, we have: $\frac{1}{t \cdot 2} \leq \frac{1}{1 \cdot 2} = 0.5$. In theory, the maximum transmission rate for each covert channel is: $v_{b_t}^{max} = 0.5$ bps.

The maximum transmission rate of covert channels built by the SDN proxy mechanism depends on C and L. In the experiments, our hardware switch can generate at most 1500 `packet-in` packets per second. Moreover, if covert information is transmitted by ARP reply packets, the available fields that can be encoded in an ARP packet are: 144 bits padding, 48 bits source MAC address in the MAC header, 48 bits sender/target MAC address in the ARP header and 32 bits sender/target IP address in the ARP header. Totally, there are 352 bits. We cannot change other fields at will. Otherwise, the ARP reply packet is broken and may fail to arrive at colluding hosts. Hence, we can calculate the maximum transmission rates of the covert channels a_s and a_t with our hardware switches in theory: $v_{a_s}^{max} = 1500 \cdot 352 = 528000$ bps and $v_{a_t}^{max} = 750$ bps. Obviously, the covert channels a_s and a_t have relatively high transmission rates compared to the covert channel b_t.

4 Information Leakage on SDN Controllers

To understand what SDN information can be obtained by an malicious application, we conduct a comprehensive investigation on five major SDN controllers, i.e., `OpenDaylight`, `ONOS`, `Floodlight`, `RYU`, and `POX`. We perform the study from two aspects. First, we summarize potential methods for SDN applications to collect information on the controller. Second, we classify the collected information. We find that an SDN application can obtain a large amount of valuable information. As the information can help develop more powerful and stealthier attacks, malicious SDN applications are well motivated to build covert channels to leak out the valuable data to a colluding host.

4.1 Collection Channels

We summarize three viable collection channels for SDN applications to collect information on the controller.

Channel I. Applications typically possess much SDN information by design to enable its designated network functionalities. For instance, a routing application knows detailed network topology and host locations to route network flows. An ARP proxy application knows host locations, IP addresses, and MAC addresses of hosts to respond to ARP request packets.

Channel II. Applications may actively retrieve information from the controller or other applications, since applications run on the same controller platform and share public system resources. Previous studies [47,50] have demonstrated that

an SDN application can get much information by reading shared data structures or common network states in controllers.

Channel III. Applications may collect some information by exploiting system misconfiguration. For instance, we investigate that one application may have access to the TLS private keys and certifications of the controller by reading configuration files (See Sect. 4.2 for details).

Table 2. SDN information and collection methods

SDN information	Collection channels	Methods
TLS keys and certifications	III	Read configuration and key files
Network security policies	I or II[a]	Query flow rules with the `FlowRuleService` object
Routing policies	I or II[a]	Query flow rules with the `FlowRuleService` object
Inter-host communication patterns	I or II[a]	Listen `packet-in` and read `HostService` object
Network topology	I or II[a]	Read `TopologyService` object
System information of controllers	III	Issue shell commands
System information of switches	I or II[a]	Read `SwitchService` object

[a]Choosing which channel depends on the application. If it possesses the information by design, the collection channel II will not be used. Otherwise, it may actively retrieve information with the channel II.

4.2 Collected SDN Information

By inspecting the implementation of real SDN controllers and conducting experiments, we identify a number of invaluable data that can be collected by an SDN application. We summarize collected SDN information in Table 2.

TLS Keys and Certifications. In SDN, the controller and switches establish secure communication channels with TLS. To build TLS connections, the controller must generate its own private keys and certifications. Although different controllers choose to store their keys and certifications in different places, we find that an application on any of the five SDN controllers can easily obtain the controller's private keys and certifications.

`POX` and `RYU` directly store the keys and certifications in **.pem* files. Due to the poor constraints on file access, we find that an application can directly obtain TLS keys and certifications by reading the **.pem* files. `FloodLight, ONOS,` and `OpenDaylight` store their keys and certifications in `JVM Keystore`, which is a repository to store key materials such as private keys, certifications, and symmetric keys. A password is required when a program wants to obtain the keys from `JVM Keystore`. It seems the TLS keys and certifications cannot be obtained without knowing the password. Unfortunately, we find that the password is written in config files of controllers, i.e., *floodlightdefault.properties* on `FloodLight`, *onos-service (a bash script)* on `ONOS`, and **-openflow-connection-config.xml* on `OpenDaylight`. As the password is saved in plaintext, an application can first read the password from configuration files and then use the password to obtain the TLS keys and certifications from `JVM Keystore`.

The TLS keys and certifications are highly sensitive. Once an application successfully leaks out the information to an end host compromised by an attacker with covert channels, the attacker may leverage it to impersonate the controllers or hijack the communication between controllers and switches.

Network Security Policies. SDN enforces various security policies to enhance network security and defend potential attacks. These policies are invisible to end users so that an attacker is hard to find possible vulnerabilities or weaknesses from the security policies. However, we find that an application on each of the five controllers can know network security policies by querying flow rules of switches with the `FlowRuleService` object. It is because the high-level security policies of network security applications in SDN are translated to the low-level flow rules in switches.

Table 3. Typical flow rules of security policies

Rules in switches	Security policies	Meaning
match=ip_src:10.0.0.1,ip_dst:10.0.0.2 actions=drop	Traffic filtering	Drop packets sent from 10.0.0.1 to 10.0.0.2
match=ip_src:10.0.0.4 actions=meter:1,output:10 meter=1 band=type:drop rate=10 Mbps	Rate limiting	Limit flow rate of host 10.0.0.4 to 10 Mbps
match=ip_src:10.0.0.3 actions=mod_ip_src:x.x.x.x,output:10[a]	MTD	Frequently change the address of a host

[a]x.x.x.x denotes the value of it is periodically changed by controllers.

Table 3 shows the typical flow rules for three types of security policies. Traffic filtering and rate limiting are widely used to mitigate a wide range of attacks, including scanning and DDoS attacks [42,53]. Moving target defense (MTD) has been proposed in SDN [25] by frequently changing the address space of a network to defend against computer worms and network scans. Table 3 shows that rules indicating traffic filtering policies and rate limiting policies can be identified by the *drop* action and the *meter* action, respectively. Rules indicating MTD policies can be identified by checking the frequent variation of IP addresses in the *mod_ip* action. Moreover, the meanings of a security policy can be known by checking detailed components of flow rules.

Routing Policies. Similar to network security policies, the high-level routing policies are translated into low-level flow rules in SDN. Thus, a malicious application can know the routing policies by querying flow rules in switches with the `FlowRuleService` object.

Inter-host Communication Patterns. Inter-host communication patterns denote what hosts are communicating with each other in the network. The patterns can reveal wide information that is useful in a multi-staged attack [40]. For example, an attacker can infer that a host may be a vital server, e.g., a local DNS server, if all hosts in the target network communicate with it. We verify

that an application can learn host communication patterns by calling the basic `HostService` object on controllers and listening `packet-in` messages. Note that `HostService` has different names on different controllers to manage the hosts. For example, `FloodLight` names it as `DeviceManager`, and `ONOS` names it as `HostSubsystem`. It can give all MAC addresses and IP addresses of hosts in the network while the destination address and source address of a flow can be extracted from the `packet_in` messages. Hence, the application can know each communication pairs of hosts.

Network Topology Information. SDN controllers maintain a global network topology view, i.e., the physical links connecting to switches and hosts in the network. The information is maintained by the basic `TopologyService` object on controllers. Our study shows that an application can obtain the network topology information by reading the `TopologyService` object. Knowing the network topology information can help an attacker to launch sophisticated DDoS attacks. For example, launching the Crossfire attack [26] requires to know the critical links in the network so that selected target servers can be effectively disconnected from the Internet once the critical links are flooded. Those critical links can be identified in SDN with the topology information and the routing policies.

System Information of Controllers. An SDN controller is typically deployed on a dedicated host with an operating system. We find that an application can obtain the controller's OS information by issuing shell commands. We find that various commands can be executed by calling *Java Runtime.exec()* in `FloodLight, ONOS,` and `OpenDaylight`, and *Python os.popen()* in `POX` and `RYU`. For instance, the application can obtain the OS type and version by calling *"uname -a"*, live TCP and UDP ports by calling *"netstat -tunlp"*, ip addresses of the host by calling *"ifconfig"*, and running processes by calling *"top"*. Such information is useful for a multi-staged attack for identifying a vulnerability in a given OS with a specific version.

System Information of Switches. By reading the `SwitchService` object, an application can know many other switch's information including switch manufacturer, hardware revision, software revision, port rate, maximal buffers, and flow table size. Knowing such information can hep an attacker to compromise a switch by exploiting the vulnerabilities of specific revisions in hardware or software [12,21]. Moreover, if the flow table size is known, an attacker can decide the minimum packet rate to overflow the flow table on a switch [18].

5 Evaluation of Covert Channels

In this section, we conduct experiments in a real SDN testbed to demonstrate the feasibility and effectiveness of our covert channels.

5.1 Experiment Setup

We build a real SDN testbed to evaluate the feasibility and effectiveness of the covert channels. The testbed consists of two commercial hardware SDN switches, `EdgeCore AS4610-54T`, and a popular open source controller, `Floodlight`. The controller runs in a server with an Intel Xeon Quad-Core CPU E5504 and 12 GB RAM. Two hosts in the testbed use `TCPReplay` to replay the real traffic trace from CAIDA [17] as the background traffic[2]. In order to evaluate different types of covert channels, we configure the testbed in two scenarios:

- The first scenario aims to evaluate the covert channels built by the SDN proxy mechanism. We deploy an ARP proxy application [4] on the controller and one host in the testbed that sends ARP requests in order to build covert channels with the application.
- The second scenario aims to evaluate the covert channels built by the SDN rule expiry mechanism. We deploy a routing [13] application on the controller. Moreover, we use `Apache HTTP Server` to build a website in a host in the testbed. The website can be visited by remote hosts on the Internet. We rent many hosts around the world to conduct experiments to demonstrate that an SDN application can successfully build covert channels with remote hosts when the hosts just visit a website in SDN. The locations of the hosts are listed in Table 4.

Table 4. Remote hosts in the experiments

Location	Host information[a]	RTT[b]
Los Angeles, USA	Intel Xeon 2.6 GHz CPU, 1 Gbps	162 ms
New York, USA	Intel Xeon 2.6 GHz CPU, 1 Gbps	215 ms
British Columbia, CA	Intel Xeon 2.6 GHz CPU, 1 Gbps	182 ms
Amsterdam, NED	Intel Xeon 2.6 GHz CPU, 1 Gbps	276 ms
Beijing, CN	Intel Xeon 1.9 GHz CPU, 1 Gbps	3 ms
ShenZhen, CN	Intel Xeon 1.9 GHz CPU, 1 Gbps	38 ms

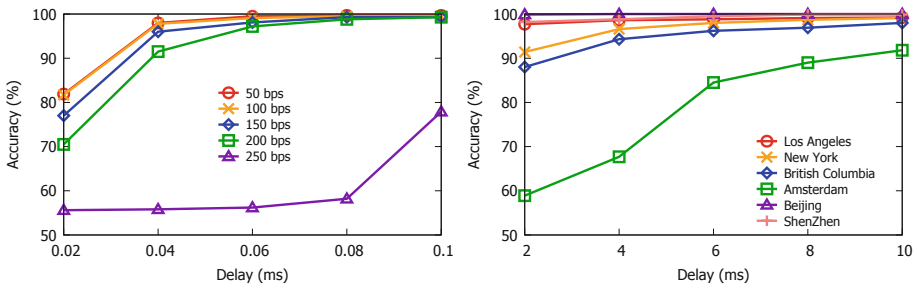
[a]The bandwidth is given by the cloud provider.
[b]The RTT is the average value between the host in the SDN testbed and the remote host on the Internet.

5.2 Experimental Results

Accuracy. Figure 3a shows the accuracy of the covert channel built by the ARP proxy mechanism. We configure the ARP proxy application to add different delays to signal "1" before sending the ARP response packet. We can see

[2] As there are so many flows in the traffic trace, the hosts randomly choose some flows to ensure that the number of flow rules generated by the flows does not exceed the flow table capacity of switches.

that the accuracy goes up with the value of the delay. Particularly, the accuracy achieves 100% when we add 0.1 ms delays to signal "1". The reason is obvious: a larger delay to signal "1" allows the covert channel to tolerate larger delay jitters when transmitting covert messages. However, the accuracy drops with the increase of the transmission rate. For instance, the accuracy reaches 80.5%, 80.2%, 76.2%, 70.7% and 55.4% with 50 bps, 100 bps, 150 bps, 200 bps, and 250 bps, respectively, when we add a 0.02 ms delay. As a higher transmission rate generates more `packet-in` and `packet-out` control messages, more CPU resources of the controller and the switches on processing the control messages are consumed and thus incur more delay jitters. Moreover, the accuracy drops significantly when transmitting covert messages with a rate of more than 200 bps. We find that it is because the switch's CPU becomes busy. Thus, we do not increase the transmission rate by generating more request packets. We do not show the accuracy of the covert channel built by the ARP proxy mechanism with the packet encoding technique in Fig. 3(a). It always achieves an accuracy of 100% with the packet encoding technique.

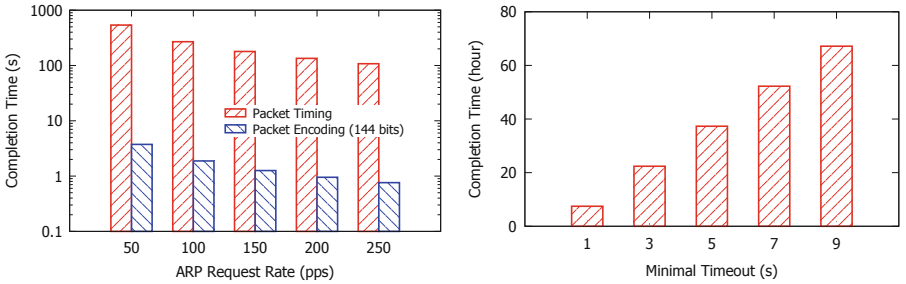


(a) ARP Reponse Timing (b) HTTP Response Timing

Fig. 3. The accuracy of covert channels with different delays on applications.

Figure 3b shows the accuracy of the covert channel built by the SDN rule expiry mechanism. The covert messages are encoded in the delays of HTTP responses. We rent many hosts in different locations on the Internet to demonstrate that covert messages can be transmitted into any remote host as long as the host can visit the website running on a host inside the target SDN. From the experimental results, we see that the accuracy increases when the application adds more delays to signal "1" before sending the control messages of rule reinstallation due to rule expiry. Particularly, the accuracy of transmitting covert messages to different locations of remote hosts all increases to more than 90% with a 10 ms delay to signal "1". The accuracy of transmitting covert messages to Amsterdam is much lower compared to other cases since the RTTs from our testbed to Amsterdam are the longest, i.e., 276 ms on average shown in Table 4. We also find that the RTTs have a relatively large variation, which can significantly affect decoding messages encoded in the delays of HTTP responses.

However, we can apply error correction codes, e.g., BCH code [24], to encode the covert messages or enlarge the delays to signal "1" so as to improve the accuracy.



(a) ARP Response Timing and Encoding (b) HTTP Response Timing

Fig. 4. The completion time of transmitting a TLS private key file of 1670 bytes.

Efficiency. Figure 4a shows the efficiency of our covert channels built by the ARP proxy mechanism when transmitting a TLS private key file of 1670 bytes. The transmission rate of the covert channels mainly depends on the ARP request rate. We can see that the completion time of transmitting the TLS key decreases with the increase of ARP request rate. The SDN application finishes transmitting the TLS key within 10 s when we encode the covert messages in the 144 bits padding of ARP response packets with an ARP request rate of 50 pps. However, the completion time can be reduced to below 1 s with an ARP request rate of 200 pps. Moreover, compared to encoding the covert messages in the padding of ARP packets, the completion time of transmitting TLS key is longer when we encode covert messages in the delays of ARP response packets. For example, the completion time is more than 100 s even with an ARP request rate of 250 pps. However, transmitting covert messages with packet timing is stealthier than that with packet encoding since there are no modifications in the ARP packets.

Figure 4b shows the efficiency of our covert channel built by the SDN rule expiry mechanism. Obviously, the timeout settings of the rule expiry mechanism totally decide the transmission rate of the covert channel. 1-bit information can be encoded in the timing of HTTP packets only when the controller reinstalls the expired flow rule. The experimental results show that the completion time of transmitting the TLS key of 1670 bytes exhibits a linear increase with the minimum value between the hard timeout and the idle timeout. Moreover, compared to the covert channels built by the SDN proxy mechanism, the completion time of transmitting the key is much longer, e.g., 7.5 h with a 1 s minimum timeout setting. Although the rate of the covert channel is much low, it allows an SDN application to stealthily transmit information to any remote host on the Internet without requiring the host inside the target SDN.

6 Possible Countermeasures

We discuss possible countermeasures to defend against the new covert channel attacks in SDN. In general, we may mitigate the covert channels in two directions, namely, preventing installing malicious SDN applications on controllers and detecting covert channels between SDN applications and colluding hosts.

Safety Check of SDN Applications. The construction of the covert channels requires malicious applications installed on SDN controllers. To avoid building covert channels, one way is to ensure that the SDN applications are benign and credible. We suggest that network administrators should install applications from trustworthy sources to avoid malicious applications. Moreover, network administrators should check the safety of an application before deploying it on controllers. If the source code of an application can be provided, static program analysis may be used to detect malicious SDN applications. For example, analyzing API calls of applications and calculating the control flow graph [31] may help to identify malicious applications. Moreover, dynamic program analysis may reveal malicious behaviors of SDN applications with sufficient test inputs, e.g., various `packet-in` control messages. However, designing an effective defense system to accurately detect malicious applications with static or dynamic program analysis is challenging and requires lots of efforts. We leave this as future work.

Control Messages Censorship on SDN Controllers. One important factor of successfully transmitting covert messages lies in that SDN applications encode messages in the timing or content of control messages. Thus, network administrators can deploy a security application that resides between the controller and other applications to inspect control messages in order to detect covert channels. Particularly, the timing between a pair of control messages that are used by an application to build covert timing channels will be relatively high. We implement such a security application in the `Floodlight` controller to time the `packet-in` and `packet-out` messages that contain ARP request and reply packets. Experimental results show that the timings between receiving the `packet-in` message and sending the `packet-out` message are typical less than 0.03 ms without covert channels. In contrast, most delays are more than 0.05 ms with covert channels, which indicates that timing control messages can be used to detect the covert timing channels. Moreover, covert storage channels that encode messages in the content of control messages can be detected when there is inconsistency in control messages. For example, suppose the ARP proxy application encodes covert messages into the 144 bits padding of ARP packets. According to the ARP protocol, the 144 bits padding should be all zeros. However, if any padding bits equal to ones, the inconsistency can be detected.

Although control messages censorship can successfully detect covert channels, it faces two challenges to be adopted in practice. The first one is on how to design a robust method that can accurately detect covert timing channels according to the timings of control packets. Different applications running on controllers with different network environments will affect the timing. For example, if the controller is busy processing network events, the delays of control

messages generated by benign applications can also be high, which easily results in a false alarm for detecting malicious applications based on the timing. The second challenge is on how to design a systematical method that can detect various inconsistency in control messages since many fields of control messages can be used to encode covert information. We leave the design of a practical censorship system as future work.

7 Related Work

In this section, we review related work in the areas of covert channels and information reconnaissance in SDN.

7.1 Covert Channels in SDN

Covert channels have been widely studied in operating systems and cloud networks [22,35,37,38]; however, there are few studies focusing on covert channels in SDN. Recently, Thimmaraju et al. [46] provide the SDN teleportation technique which allows malicious switches to build covert channels for hidden communication with each other. SDN teleportation mainly exploits three SDN functionalities: flow re-configurations, switch identification, and out-of-band forwarding. It allows malicious switches to generate spoofing control messages to controllers, which leads controllers to send extra control messages to other switches. Krösche et al. [30] further studies the SDN teleportation technique with switch identification. They build the state machine of switch identification and model it in terms of time delays to build a covert timing channel with a high accuracy. Different from the above two studies that build covert channels between switches and switches, our work exploits the unique SDN proxy mechanism and rule expiry mechanism to build covert channels between SDN controllers and remote hosts. Our covert channels can be used to transmit information from SDN controllers to end hosts while they can bypass physical network isolation and security policies.

7.2 Information Reconnaissance in SDN

Information reconnaissance in SDN has been widely studied. Shin et al. [43] and Cui et al. [20] study the feasibility of inferring whether a network adopts SDN by measuring delays of pings. Köti et al. [28] infer if there are aggregated flow rules for TCP flows by timing TCP setup delays. John et al. [45] present a complicated inference attack that learns host communications and access control lists in SDN by timing the control plane's execution. Achleitner et al. [15] design SDNMap that reveals the detailed composition of flow rules by sending probe packets of various network protocols. Liu et al. [36] build a Markov model of SDN switches, which allows an attacker to choose the best probes to infer whether a flow occurred recently in the network. Previous work mainly depends on the timing-based side channel or protocol feature to infer some SDN information. In contrast, our work focuses on building covert channels to leak out many types of valuable SDN information from controllers to end hosts.

8 Conclusion

In this work, we study how a malicious SDN application can transmit information from controllers to end hosts while bypassing network security policies and physical network isolation. For that purpose, we design two types of covert channels with the basic SDN proxy and rule expiry mechanisms, respectively. These covert channels can transmit information either to a host in the target SDN at a high rate or to a host on the Internet at a low rate. We demonstrate the feasibility of the covert channels with experiments in a real SDN testbed. In addition, we make a comprehensive study on five major SDN controllers to understand what information an SDN application can leak out from controllers. Finally, we discuss possible countermeasures to mitigate the covert channels.

Acknowledgment. The research is partially supported by the National Natural Science Foundation of China (NSFC) under Grant 61832013, 61625203, 61572278 and U1736209, the National Key R&D Program of China under Grant 2017YFB0803202, and the NSF grants IIP-1266147 and CNS-1822094.

References

1. Access Control in ONOS Controller. https://wiki.onosproject.org/display/ONOS/Access+Control+Based+on+DHCP
2. Firewall Application in Floodlight Controller. https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343616/Firewall
3. Floodlight DHCP Proxy Service. https://github.com/floodlight/floodlight/tree/master/src/main/java/net/floodlightcontroller/dhcpserver
4. Floodlight ProxyARP. https://github.com/mbredel/floodlight-proxyarp
5. Manchester Code. https://en.wikipedia.org/wiki/Manchester_code
6. Microsoft Azure and Software Defined Networking. https://docs.microsoft.com/en-us/windows-server/networking/sdn/azure_and_sdn/
7. ONOS Neighbour Resolution Service for ARP and NDP Proxy. https://wiki.onosproject.org/display/ONOS/Neighbour+Resolution+Service
8. ONOS ProxyARP. https://github.com/opennetworkinglab/onos/blob/master/apps/proxyarp/src/main/java/org/onosproject/proxyarp/DefaultProxyArp.java
9. OpenDayLight ARP Proxy Service. https://github.com/opendaylight/honeycomb-vbd/blob/master/api/src/main/yang/proxy-arp
10. OpenDayLight Neutron DHCP Proxy Service. https://docs.opendaylight.org/en/stable-nitrogen/submodules/netvirt/docs/specs/neutron-port-for-dhcp-service.html
11. OpenFlow Specification v1.5.1. https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf
12. OpenvSwitch: Products and Vulnerabilities. https://www.cvedetails.com/vendor/12098/Openvswitch.html
13. Routing Application on Floodlight. https://github.com/floodlight/floodlight/tree/master/src/main/java/net/floodlightcontroller/routing/
14. Abdelltif, A.A., et al.: SDN-based load balancing service for cloud servers. IEEE Commun. Mag. **56**(8), 106–111 (2018)

15. Achleitner, S., et al.: Adversarial network forensics in software defined networking. In: ACM SOSR, pp. 8–20 (2017)
16. Braun, W., Menth, M.: Software-defined networking using openflow: protocols, applications and architectural design choices. Futur. Internet **6**(2), 302–336 (2014)
17. CAIDA Passive Monitor: Chicago B: http://www.caida.org/data/passive/trace_stats/chicago-B/2015/?monitor=20150219-130000.UTC
18. Cao, J., Xu, M., Li, Q., Sun, K., Yang, Y., Zheng, J.: Disrupting SDN via the data plane: a low-rate flow table overflow attack. In: Lin, X., Ghorbani, A., Ren, K., Zhu, S., Zhang, A. (eds.) SecureComm 2017. LNICST, vol. 238, pp. 356–376. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78813-5_18
19. Chiang, S.-H., et al.: Online multicast traffic engineering for software-defined networks. In: IEEE INFOCOM, pp. 414–422 (2018)
20. Cui, H., et al.: On the fingerprinting of software-defined networks. IEEE TIFS **11**(10), 2160–2173 (2016)
21. Dhawan, M., et al.: Sphinx: detecting security attacks in software-defined networks. In: NDSS, vol. 15, pp. 8–11 (2015)
22. Gras, B., et al.: Translation leak-aside buffer: defeating cache side-channel protections with TLB attacks. In: USENIX Security, pp. 955–972 (2018)
23. Hizver, J.: Taxonomic modeling of security threats in software defined networking. In: BlackHat Conference, pp. 1–16 (2015)
24. Hocquenghem, A.: Codes correcteurs d'erreurs. Chiffres **2**(2), 147–156 (1959)
25. Jafarian, J.H., et al.: OpenFlow random host mutation: transparent moving target defense using software defined networking. In: ACM HotSDN, pp. 127–132 (2012)
26. Kang, M.S., et al.: The crossfire attack. In: IEEE Symposium on Security and Privacy, pp. 127–141 (2013)
27. Katta, N., et al.: Infinite cacheflow in software-defined networks. In: ACM HotSDN, pp. 175–180 (2014)
28. Klöti, R., et al.: OpenFlow: a security analysis. In: IEEE ICNP, pp. 1–6 (2013)
29. Kreutz, D., et al.: Software-defined networking: a comprehensive survey. Proc. IEEE **103**(1), 14–76 (2015)
30. Krösche, R., et al.: I DPID it my way! A covert timing channel in software-defined networks. In: IFIP Networking (2018)
31. Lam, P., et al.: The soot framework for java program analysis: a retrospective. In: CETUS 2011, vol. 15, p. 35 (2011)
32. Lee, S., et al.: The smaller, the shrewder: a simple malicious application can kill an entire SDN environment. In: ACM SDN-NFV Security, pp. 23–28 (2016)
33. Li, H., et al.: vNIDS: towards elastic security with safe and efficient virtualization of network intrusion detection systems. In: ACM CCS, pp. 17–34 (2018)
34. Lin, Y.-D., et al.: OFBench: performance test suite on OpenFlow switches. IEEE Syst. J. **12**(3), 2949–2959 (2018)
35. Lipp, M., et al.: Meltdown: reading kernel memory from user space. In: USENIX Security, pp. 973–990 (2018)
36. Liu, S., et al.: Flow reconnaissance via timing attacks on SDN switches. In: IEEE ICDCS, pp. 196–206 (2017)
37. Maurice, C., et al.: Hello from the other side: SSH over robust cache covert channels in the cloud. In: NDSS (2017)
38. Moon, S.-J., et al.: Nomad: mitigating arbitrary cloud side channels via provider-assisted migration. In: ACM CCS, pp. 1595–1606 (2015)
39. Narten, T.: Neighbor Discovery for IP version 6. RFC 2461 (1998)
40. Ou, X., et al.: A scalable approach to attack graph generation. In: ACM CCS, pp. 336–345 (2006)

41. Porras, P.A., et al.: Securing the software defined network control layer. In: NDSS (2015)
42. Rossow, C.: Amplification hell: revisiting network protocols for DDOS abuse. In: NDSS (2014)
43. Shin, S., Gu, G.: Attacking software-defined networks: a first feasibility study. In: ACM HotSDN, pp. 165–166 (2013)
44. Shin, S., et al.: Rosemary: a robust, secure, and high-performance network operating system. In: ACM CCS, pp. 78–89 (2014)
45. Sonchack, J., et al.: Timing-based reconnaissance and defense in software-defined networks. In: IEEE ACSAC, pp. 89–100 (2016)
46. Thimmaraju, K., et al.: Outsmarting network security with SDN teleportation. In: IEEE EuroS&P, pp. 563–578 (2017)
47. Ujcich, B.E., et al.: Cross-app poisoning in software-defined networking. In: ACM CCS (2018)
48. Wang, H., et al.: Towards fine-grained network security forensics and diagnosis in the SDN era. In: ACM CCS, pp. 3–16 (2018)
49. Wen, X., et al.: SDNshield: reconciliating configurable application permissions for SDN app markets. In: IEEE/IFIP DSN, pp. 121–132 (2016)
50. Xu, L., et al.: Attacking the brain: races in the SDN control plane. In: USENIX Security, pp. 451–468 (2017)
51. Yoon, C., Lee, S.: Attacking SDN infrastructure: are we ready for the next-gen networking? In: BlackHat-USA (2016)
52. Yoon, C., et al.: A security-mode for carrier-grade SDN controllers. In: ACM ACSAC. pp. 461–473 (2017)
53. Zheng, J., et al.: Realtime DDoS defense using COTS SDN switches via adaptive correlation analysis. IEEE TIFS **13**(7), 1838–1853 (2018)