Adversarial Network Forensics in Software Defined Networking

Stefan Achleitner, Thomas La Porta, Trent Jaeger, Patrick McDaniel
Computer Science and Engineering, Pennsylvania State University
University Park, PA 16802
{sachleitner,tlp,tjaeger,mcdaniel}@cse.psu.edu

ABSTRACT

Software Defined Networking (SDN), and its popular implementation OpenFlow, represent the foundation for the design and implementation of modern networks. The essential part of an SDN-based network are flow rules that enable network elements to steer and control the traffic and deploy policy enforcement points with a fine granularity at any entry-point in a network. Such applications, implemented with the usage of OpenFlow rules, are already integral components of widely used SDN controllers such as *Floodlight* or *OpenDayLight*. The implementation details of network policies are reflected in the composition of flow rules and leakage of such information provides adversaries with a significant attack advantage such as bypassing *Access Control Lists (ACL)*, reconstructing the resource distribution of *Load Balancers* or revealing of *Moving Target Defense* techniques.

In this paper we introduce a new attack vector on SDN by showing how the detailed composition of flow rules can be reconstructed by network users without any prior knowledge of the SDN controller or its architecture. To our best knowledge, in SDN, such reconnaissance techniques have not been considered so far. We introduce *SDNMap*, an open-source scanner that is able to accurately reconstruct the detailed composition of flow rules by performing active probing and listening to the network traffic. We demonstrate in a number of real-world SDN applications that this ability provides adversaries with a significant attack advantage and discuss ways to prevent the introduced reconnaissance techniques. Our SDNMap scanner is able to reconstruct flow rules between network endpoints with an accuracy of over 96%.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSR'17, Santa Clara, CA
© 2017 ACM. © 2017 ACM. ISBN 978-1-4503-4947-5/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3050220.3050223

CCS CONCEPTS

•**Security and privacy →Network security;** *Security protocols; Domain-specific security and privacy architectures;*

KEYWORDS

OpenFlow rule reconstruction, Attacks on SDN

ACM Reference format:
Stefan Achleitner, Thomas La Porta, Trent Jaeger, Patrick McDaniel. 2017. Adversarial Network Forensics
in Software Defined Networking. In *Proceedings of ACM Symposium on SDN Research, Santa Clara, CA, April 3–4, 2017 (SOSR'17),* 13 pages.
DOI: http://dx.doi.org/10.1145/3050220.3050223

1 INTRODUCTION

Software Defined Networking (SDN) has received a significant amount of attention in both the academic research community and industry due to its potential for the implementation of dynamic and flexible networked systems. The essential part of SDN-based networks and applications are flow rules, deployed on SDN-enabled switches, to control the network traffic. The construction of flow rules is defined by protocols, such as OpenFlow [9], which is widely used in the major open-source SDN controllers such as OpenDayLight or Floodlight. In this paper we study OpenFlow-based SDNs.

In comparison to legacy networks, SDN enables a central controller to dynamically program the data plane and implement applications and network policies with a fine granularity of information to steer and control traffic. In legacy networks policy enforcement techniques such as firewalls or ACLs are deployed at the edge of a network, in contrast OpenFlow-enabled networks give a deeper level of granularity to SDN controllers and enables them to place enforcement points at any entry point in a network [7, 22].

Since the internals of SDN-enabled switches are not accessible to network users, they appear as a black-box and the deployed system of flow rules is assumed to be invisible to users. This central assumption is especially important for the deployment of SDN-based security-aware policies.

In this paper we infer the composition details of OpenFlow rules on the data plane, independent of the controller platform, by developing traffic analysis techniques that are

able to gather high granularity information about SDN-based packet forwarding. Our approach is able to gather enough information to infer the detailed composition of rules, including matching and action fields along with their associated field values. To our best knowledge, such reconnaissance techniques have not been considered so far as a potential attack vector in the context of SDN-based networks. Using our approach, we demonstrate that by inferring flow rule details, adversaries can reconstruct deployed network policies and use this knowledge for crafting attack traffic to bypass ACLs, map load balancing strategies or defeat moving target defense implementations to name a few examples.

Existing adversarial probing techniques used in firewall fingerprinting [18, 25, 27, 30], collect a coarser granularity of packet information, limited to port numbers, IP addresses and protocols, used to filter packets at policy enforcement points. In contrast, OpenFlow considers a significantly larger range of information over multiple network layers, to identify packets. Additionally, dynamic packet manipulation actions executed by flow rules and required for the implementation of novel security mechanisms, such as moving target defense, are neglected by existing probing techniques. Inferring the details of SDN-based policy enforcement points deployed with a fine granularity requires advanced probing techniques to collect information ranging over multiple network layers, contrary to traditional techniques targeting policies located at the edge of a network.

We introduce scanning methods that are able to retrieve the required information to fully and accurately reconstruct flow rules. Following the principle of "low and slow", our probing techniques aim to reduce the probes to one packet per second on average for reconstructing flow rules from the scanning source to a destination host. In Section 5 we show that this approach makes SDNMap stealthy by staying below the traffic threshold considered in recent SDN defense techniques [20]. This new attack vector on analyzing the precise forwarding policy of SDN switches, gives an adversary detailed knowledge of what packet and meta information is used to match packets and which actions are applied.

We also discuss the effects of current defense systems, focused on SDN, on our attack approach and list improvements to prevent the re-construction of flow rules. The following is a summary of our contributions:

- ***Development of rule reconstruction techniques***
  We develop network probing techniques to infer the composition of OpenFlow rules that implement specific network policies on the data plane. By applying these probing techniques in a set of scanning steps our approach can retrieve a fine enough granularity of information to determine the construction details of flow rules, which can be used to craft adversarial network traffic.
- ***Analyzing new attack vectors in SDN***
  We evaluate the developed probing techniques by applying them on various real-world application scenarios and show that knowledge about the construction details of OpenFlow rules gives adversaries a significant advantage for the execution of targeted attacks in SDN.

Based on the introduced adversarial probing techniques we implement SDNMap, an open-source tool available at [17], that can be used as a scanner for the reconstruction of OpenFlow rules on the data plane. We show that SDNMap is able to reconstruct flow rules to a target host with an accuracy of over 96%.

2 MOTIVATION

The dynamic programmability of switches is a characteristic distinguishing SDN from traditional networks. Policies (e.g. ACL's, user identification, firewalls), entire applications (e.g. load balancers) or novel defense mechanisms (e.g. moving target defense) can be implemented with flow rules defined by protocols such as OpenFlow [9]. Typically, SDN controllers follow either a reactive or a proactive approach for flow rule deployment. For the analysis of reconstructing flow rules on the data-plane we consider both approaches in this paper. The definition of flow rules is based on fields that analyze, control and manipulate traffic with a fine granularity, extending the precision and flexibility of traditional network policies (e.g. firewalls).

The internal functionality of SDN-enabled switches, determined by the deployed flow rules, is assumed to be invisible to network users, which is especially critical for applications with a security purpose such as access control, or resource distribution as achieved by load balancers. Construction details of rules which control the traffic between network endpoints, reveals the internal functionality of SDN-enabled switches and give adversaries an advantage in the identification of targets and development of a battle plan for the execution of attacks.

2.1 Problem statement

The challenge in the development of rule-reconstruction techniques is to infer the internal functionality of an OpenFlow enabled network element by determining which packet information was used to make a matching decision and which actions are applied on matching packets. Since network users are not able to gain access to the SDN-switch internals, this information has to be inferred by transmitting probing packets. We consider a source host which aims to determine the detailed composition of flow rules and sends probing packets to a set of target hosts, which are defined by a list of IP addresses. A scanner aiming to infer the details of OpenFlow rules has to make the target hosts reply in a way that leaks enough information to reveal the used flow rule fields to make a matching decision and the applied rule actions.

Table 1: OpenFlow fields reconstructed with different probing techniques

OpenFlow field	Type[2]	SDNMap	[a]FWFP	[b]FPSDN	[c]Firewalk	[d]Firecracker	[e]Lumeta
Ingress port (SIP) (used/not used)	M	✓					
MAC destination address (HWd)	M	✓					
MAC source address (HWs)	M	✓					
Ethernet type (PT)	M	✓(ARP, IP)					
IPv4 protocol (PT)	M	✓(ICMP, TCP, UDP)	✓(TCP, UDP)		✓(TCP, UDP)	✓(TCP, UDP)	✓(TCP, UDP)
IPv4 source address (IPs)[1]	M	✓				✓	✓
IPv4 destination address (IPd)[1]	M	✓			✓	✓	✓
TCP/UDP source port (POs)	M	✓	✓		✓	✓	
TCP/UDP destination port (POd)	M	✓			✓	✓	✓
Egress action (FA) (forward/drop)	A	✓	✓	✓	✓	✓	✓
Modify IPv4 src address (rIPs)	A	✓					
Modify IPv4 dst address (rIPd)	A	✓					

Existing probing techniques, such as used in firewall fingerprinting, can not be directly applied in SDN since firewalls are usually deployed on the edge of the network and only collect a limited set of information used for packet filtering. In contrast OpenFlow enabled network elements have a significantly larger set of fields to identify packets and apply flow actions. Furthermore, network policies implemented with OpenFlow can be deployed with a higher granularity, at any entry point of the network.

To determine the construction details of OpenFlow rules, a finer granularity of information is required to infer the matching as well as the action part of a flow rule. We overcome this challenge by implementing scanning techniques which retrieve a fine granularity of information to determine the flow rule details controlling traffic between hosts.

Since the main focus of this work is to map the internals of an OpenFlow-enabled switch, we consider pure SDN-based networks. We do not consider hybrid networks (SDN and legacy combined) in this work.

2.2 Comparison of probing techniques

To set our approach apart from existing scanning techniques aiming to infer network security policies, we analyze which information existing probing techniques are able to infer. In Table 1 we show that existing techniques are at most collecting half of the information required to reconstruct the core list of the OpenFlow [9] protocol fields, as well as address rewriting techniques frequently used in SDN applications. Based on the analysis presented in this paper, we provide an implementation of our rule-reconstruction scanner SDNMap [17] which is easy to execute and provides a significant advantage to adversaries for the planning and execution of targeted attacks in SDN-based networks, as we demonstrate in Section 4. We plan to further extend SDNMap in our future work to reconstruct an extended part of the OpenFlow protocol. In the following we briefly discuss the existing comparison techniques listed in Table 1.

[a] *Firewall Fingerprinting (INFOCOM 2012) [25]*: A probing technique is presented using either a sequence of TCP or UDP probing packets with fixed header and varying source port numbers, which evaluates the packet classification algorithm used and its performance.

[b] *Fingerprinting Software-defined Networks (ICNP 2015) [19]*: The authors of this work propose a technique of network fingerprinting to determine architectural details of an SDN. As shown in Table 1, this technique is only able to determine if a packet is forwarded or not, but is not able to retrieve a fine granularity of information to infer details of flow rule construction.

[c] *Firewalk Firewall scanner (Open-source software) [4]*, [d] *Firecracker: A framework for inferring firewall policies using smart probing (ICNP 2007) [27]*, [e] *Architecting the Lumeta Firewall Analyzer (USENIX 2001) [30]*: Firewalk, Firecracker and Lumeta are tools and papers focused on reconstructing firewall policies. While not directly applicable to SDN, the techniques are able to determine if certain IP addresses and port numbers are allowed or denied. Such information could be translated into SDN rules used to implement Firewalls or ACL's. As shown in Table 1, firewall reconnaissance techniques are able to identify about half of the information SDNMap is retrieving.

2.3 Threat model

In our threat model we consider a user who has access to a computer connected to a SDN-based network and is able to run SDNMap with root privileges. The user has no prior knowledge of the network architecture or any privileges of a network administrator or operator. We do not assume that the SDN controller, other SDN elements or the probed destination hosts are compromised by a malware or any other adversarial user.

Techniques to prevent network scanning often depend on monitoring packet throughput or blocking of certain protocols. The scanning procedure in SDNMap does not depend on a single protocol to perform the required steps; TCP or ICMP can be used interchangeably as we discuss in Section 3. SDNMap does not perform actions such as link fabrication

[1] OpenFlow also defines the usage of IPv6 addresses. We do not consider IPv6 in this version of SDNMap, but are planning to include IPv6 in future versions of SDNMap.
[2] M=Matching, A=Action

or creating a fake topology targeted by recent SDN-specific defense approaches ([20, 21]), but aims to exploit the details of existing flow rules and network policies. Following the principle of "low and slow" SDNMap transmits less than one packet per second during the scanning process on average, depending on the user settings. We evaluate the effects of current SDN-specific defense techniques in Section 5.

3 RULE RECONSTRUCTION

To infer the details of an OpenFlow rule we transmit probing packets with the actual source host header information and compare the received response to probing packets that use spoofed values for the evaluated header field. To overcome the challenge of inferring matching and action parts defined in OpenFlow rules we exploit certain features in existing network protocols.

3.1 Rule reconstruction overview

To scan a SDN-based network and reconstruct OpenFlow rules, we assume a user can detect if a network is SDN-based using previously published techniques such as [19, 28].

We seek to reconstruct rules using only external entities, i.e. hosts that are connected to an SDN-based network. We do not assume that the controller or network elements, i.e. SDN switches, are compromised. The challenge of reconstructing OpenFlow rules is to craft a sequence of probes that cause sufficient information to be leaked so the details of rules can be determined. Our approach aims to craft the required sequence of probes to network endpoints with very low overhead.

We introduce techniques in this paper to infer the matching and action fields as well as their associated values. The scanning process to determine the composition of flow rules is performed on the data plane, where OpenFlow specific rules are reactively or proactively deployed. This makes our approach independent of specific controller platforms.

In OpenFlow, rule fields have a specific value assigned and evaluate arriving packets for that specific value. Multiple values, e.g. a list of port numbers, cannot be checked in a single OpenFlow rule as defined in the protocol specification [9]. With the exception of IP addresses which can be evaluated by network prefix (e.g. 10.0.0.0/16).

In the following we present an overview of these scanning steps along with the number of probing packets required by each step. The number of probing packets are required to infer the flow rules from the scanning host to a destination host, that is specified by the SDNMap user. SDNMap allows to scan single or multiple hosts by specifying a range of IP addresses to probe. To resolve the corresponding MAC addresses for the supplied range of IP addresses SDNMap transmits ARP request packets as we discuss in the following. In case multiple hosts are scanned, the introduced scanning steps have to be executed multiple times.

- **Host reachability (2 probing packets)**
  As an initial scanning step, SDNMap transmits an ARP request packet and a probing packet (TCP or ICMP) with the real packet header field values of the probing source host to the destination host. With this scanning step, we evaluate if the flow rules controlling the traffic between the source and destination host forward benign packets with legitimate header information.
- **MAC addresses (4 probing packets)**
  In this scanning step SDNMap sends probing packets to infer if the matching part of the flow rule forwarding packets between the scanning and destination host, considers specific MAC source and destination addresses to identify a packet. By executing this scanning step we are able to infer if the fields $match{:}MAC_{src}{=}HWs$, where HWs is the scanning host's actual MAC address and the field $match{:}MAC_{dst}{=}HWd$, where HWd is the destination host's actual MAC address, are used in the matching part of the flow rule. We discuss this step in Section 3.2.
- **IP addresses (3 probing packets)**
  Following a similar approach as inferring MAC address fields, in this scanning step we send probing packets to determine if specific IP addresses are checked in the matching part of an OpenFlow rule. By executing this scanning step we are able to infer if the field $match{:}IP_{src}{=}IPs$, where IPs is the actual IP address of the scanning host, and $match{:}IP_{dst}{=}IPd$, where IPd is the actual IP address of the destination host are used as matching criteria in a flow rule. We discuss this step in detail in Section 3.3.
- **Protocols and ports ($2 + 2p^2$ probing packets)**
  To evaluate if a flow rule is matching packets for specific protocols, SDNMap sends probing packets of the type ARP, ICMP, TCP and UDP to the destination host and analyzes the received responses. For the TCP and UDP protocol, SDNMap also evaluates if a list of user defined port numbers is checked in the matching part of a flow rule. This scanning step determines if the protocol field PT, where $PT \in \{ARP, ICMP, TCP, UDP\}$ is considered in a flow rule. If $PT = \{TCP \text{ or } UDP\}$ SDNMap also determines if the field $tp_{src} = POs$ is matching packets for a specific source port number POs and if $tp_{dst} = POd$ is matching packets for a specific destination port number POd. For this step a user specified list p of port numbers is considered for the scanning procedure. We discuss this step in detail in Section 3.4.
- **Ingress port (2 probing packets)**
  The OpenFlow field ingress port can be seen as meta information which is only available once a packet arrives at the switch. SDNMap is not able to determine the actual ingress port number, but can infer if the field $match{:}in\_port{=}SIP$ is used in the matching part of a flow rule. The value

SIP is replaced with the placeholder *#IN_PORT*. We determine this value by impersonating another host in the same sub-network as we will discuss in Section 3.5.

- **IP address rewriting action (1 probing packet)**
  The action part of an OpenFlow rule is executed if a packet fulfills all matching criteria of a flow rule. For inferring if an OpenFlow rule is performing IP address rewriting actions, SDNMap transmits a UDP packet which triggers an ICMP error message at the destination host that will reveal the actual IP addresses received by the destination. With this scanning step we are able to determine the fields *actions:set_IP_src =rIPs* for rewriting the IP source address and *actions:set_IP_dst=rIPd* for rewriting the IP destination address, where *rIPs* is the value overwriting the original IP source address and *rIPd* the value overwriting the original IP destination address. We will discuss the details of this scanning step in Section 3.6.
- **Forwarding action**
  The forwarding action in a flow rule either transmits a packet to a specific switch port or drops the packet. Here, our approach is able to infer which action, forwarding or dropping of a matching packet, is used. SDNMap concludes this action based on the received response packets from the previously executed scanning steps as discussed.

In the following we will discuss the technical details of the introduced scanning steps.

3.2 Scanning step: MAC addresses

In this scanning step we infer if the flow rule fields $match{:}MAC_{src}$ $=HWs$ and $match{:}MAC_{dst}=HWd$ evaluate a packet for specific MAC source (HWs) and destination (HWd) addresses.

To determine the usage of MAC address fields and the specific values of HWs and HWd we exchange packets as shown in Figure 1. For this scanning step, SDNMap generates probing packets, by using either TCP or ICMP, with a spoofed MAC source address. To generate a spoofed MAC source address we use the first three octets of its current physical host which is also known as the *Organizationally Unique Identifier (OUI)*, and select the last three octets of the spoofed MAC address randomly. This procedure will generate a valid MAC address, with a high likelihood that no other host in the network has the same MAC address.

To generate a reply packet to the transmitted probe of our SDNMap scanner, the destination host will not use the received source MAC address of the probing packet, but will lookup the corresponding MAC address of the received source IP address in its local ARP cache as explained in RFC 826 [16]. If an entry is present, the host will generate a probe reply message with the MAC destination address from its local ARP cache and send it back to the source host. If an entry for the IP in its local ARP cache is not present, the destination host will send out an ARP request packet to resolve the received source IP address in our probing packet. If a reply for the ARP request was received, the host updates the entry in its local ARP cache and will send the probe reply packet back to the source host.

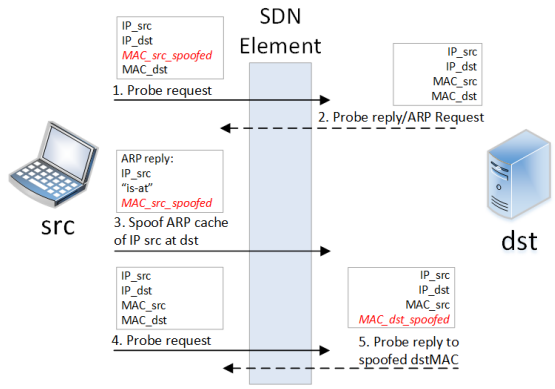

Figure 1: Steps to determine MAC address fields

Upon receiving a probe reply or ARP request packet, our SDNMap scanner can conclude that the MAC source address was not used for transmitting the probe packet to the destination host since it was spoofed. If no reply packet to the probing message was received, the procedure can be repeated to eliminate other causes for not receiving a reply packet, such as packet loss. If no probe reply or ARP request packet is received after a specified waiting threshold, it can be assumed that the source MAC address was used as a matching criteria to deliver the packet to the destination host. The explained process is visualized with messages *1* and *2* in Figure 1.

To determine if a flow rule uses the MAC destination address, the SDNMap scanner does not directly send probes with spoofed destination MAC addresses since such a packet could be misrouted, but makes use of a well know vulnerability of the ARP protocol, called ARP poisoning. SDNMap uses this vulnerability to make the destination host reply to a probing packet with a MAC address of our choice to reveal the usage of the MAC destination address field in an OpenFlow rule. We assume that bidirectional rules between a source and destination host are constructed in a similar way. As defined in RFC 826 [16] and further discussed in RFC 5227 [13], hosts correctly implementing the ARP protocol will accept ARP reply packets, even if they were not preceded by an ARP request packet. We are aware that there are methods to prevent ARP poisoning in networks. Nevertheless, ARP poisoning is alive and a security threat since defense methods are often hard to deploy in real-world scenarios. Static configurations of ARP caches are very hard to maintain as discussed in [21] and not recommended by major network vendors [3]. Further, it has recently been shown in [20] that major SDN controllers (Floodlight, OpenDayLight, POX) are

vulnerable to ARP poisoning since traditional defense techniques cannot trivially be extended to SDN-based networks.

As a reconnaissance tool, SDNMap uses ARP cache poisoning in a slightly different way than it is traditionally exploited by attackers. As shown in messages *3*, *4* and *5* in Figure 1, we first send an ARP reply message to the destination host that will update its cache entry for the IP address of our probing host to the spoofed MAC address. After that we send a correct probe request packet to the destination. Upon arrival of our probe, the destination host will now lookup the corresponding MAC address to the received IP source address, which is now the spoofed MAC and will send a probe reply message back to the requester with the spoofed MAC address as the layer 2 destination address. If the probe reply packet is received at the SDNMap host our SDNMap scanner will conclude that the destination MAC address was not used as a criteria in the flow rule matching process to deliver the packet.

If no reply packet to the probe request is received after a defined timeout, SDNMap will conclude that the correct MAC destination address was used by the flow rule to deliver the packet. To restore the correct ARP cache entry for the probing host's IP address at the destination, SDNMap will send an ARP reply packet to the destination with the correct addresses (not shown).

Two steps, as described, are required to determine if layer 2 information is used for the forwarding of packets, since it is possible that a flow rule uses source MAC and/or destination MAC address. If we would only use messages *3*, *4* and *5* shown in Figure 1 and the flow rules only use the destination MAC address field, SDNMap would not be able to find out if the source MAC address field is used or not.

3.3 Scanning step: IP addresses

To determine the usage of IP addresses, this scanning step infers if the fields $match{:}IP_{src}{=}IPs$ and $match{:}IP_{dst}{=}IPd$ are used in a flow rule to match a packet for specific IP source (*IPs*) and destination (*IPd*) addresses. In Figure 2 we give an overview of the probing steps our SDNMap scanner performs to determine the values *IPs* and *IPd* and the usage of their associated fields in the matching part of a flow rule.

To reconstruct the usage of IP addresses, we generate either a TCP or ICMP probing packet and set the source IP address to a spoofed address as shown in packet *1* in Figure 2. For the generation of spoofed IP addresses, SDNMap uses the network prefix of its host machine and selects the host number in a random way. In addition multiple probing packets with different spoofed IP source addresses can be sent to the destination host, to ensure that a packet with an unseen IP source address will arrive.

In case SDNMap is scanning a host within the same subnetwork, the delivered packet will generate an entry in the destination host's ARP cache. This will trigger the transmission of an ARP request packet which the destination host will broadcast into the network to determine the corresponding MAC address to the received spoofed IP address.

If a host in a distant sub-network is probed, the network gateway will receive the probing packet and broadcast an ARP request to the originating sub-network, to determine the senders MAC address.

In both cases, receiving an ARP request, SDNMap can conclude that the packet with the spoofed IP address was received and that the source IP address is not used by the SDN network element for the forwarding of packets to the destination host.

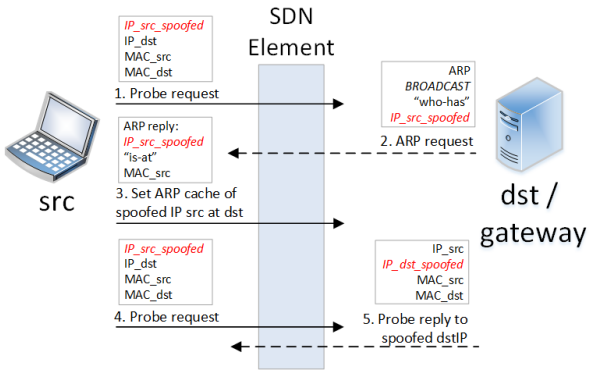

Figure 2: Steps to determine IP address fields

If no ARP request packet from the destination host was received, the procedure is repeated to exclude delivery failures like lost packets on the communication path. If no ARP request was received after multiple attempts, SDNMap will conclude that the flow rule is matching packets for a specific IP source address and therefore a packet with a spoofed IP source address will not be delivered to the destination host.

Upon the reception of an ARP request packet for the spoofed IP address used in the first probing message, SDNMap will send an ARP reply message telling the destination that the spoofed IP address is at the SDNMap's host MAC address. In message *4*, SDNMap will now re-send a probing packet with the spoofed IP source address to the destination. This will trigger the generation of a reply packet and makes the destination host send a packet with the a spoofed destination IP address back to the probing host which enables us to infer the usage of the IP destination field.

If a probing reply packet, shown as *5* in Figure 2, is received by our SDNMap scanner and since we assume that most bidirectional rules are constructed in a similar way, it will conclude that the IP destination is not used as part of the flow rule matching field. If no reply packet was received by SDNMap after multiple attempts, our software will conclude that the correct IP destination address is required by the matching process and is therefore part of the flow rule.

3.4 Scanning step: Protocols and ports

In this scanning step, our approach determines if flow rules are using OpenFlow fields to match packets for specific protocols and use the fields $tp_{src} = POs$ and $tp_{dst} = POd$ to match packets for specific source *POs* and destination *POd* ports. SDNMap evaluates rules for the protocols ARP, IP, ICMP, TCP and UDP.

SDNMap determines if ARP packets are forwarded by the SDN controller, by sending an ARP request packet to the specified destination IP in the initial scanning step as discussed in Section 3.1. To determine if ICMP packets are forwarded we send ICMP *Echo Request (type 8 code 0)* packets and listen for *Echo Reply (type 0 code 0)* packets.

If the SDN controller generates flow rules matching packets for protocols such as TCP or UDP, the accepted source and destination ports, can also be part of the matching criteria used in flow rules. Since there is a huge number of source and destination port combinations, a user can pass a list of ports p to be checked as an argument when SDNMap is started. Our scanner takes the list of supplied port numbers and checks all possible source and destination port combinations by sending probing packets. The matching part of an OpenFlow rule is able to evaluate packets for specific source and destination port numbers. By supplying a list of port numbers p, SDNMap is able to evaluate if a flow rule between the source and destination host is matching packets for the specified ports. If no list of specific ports to evaluate is supplied by the user, SDNMap will not determine the exact allowed port numbers, but will select a source and destination port in p randomly between the ranges *35000-65000*, to evaluate if a flow rule is matching for the TCP or UDP protocol.

To determine if flow rules to the probed destination host are matching packets for the TCP protocol, we send a TCP SYN packet to a port at the destination host. In case a TCP probing packet is sent to an open port, the destination host will send a TCP packet with the ACK flag back to the probing host. If the TCP packet is sent to a closed port, the destination will answer with a TCP RST packet as specified in RFC 793 [15]. By receiving a TCP packet with the ACK or RST flag set, we can derive that TCP packets are forwarded by the flow rule controlling traffic to the destination host.

It is also possible to evaluate the reception of TCP packets at the destination host by overhearing ARP request packets. Therefore, SDNMap transmits TCP packets with spoofed source IP addresses to the destination host. The reception of a packet with a spoofed IP address at the destination will trigger the transmission of an ARP request packet by the destination. Overhearing an ARP request packet by SDNMap also confirms the reception of the TCP probing packet.

Determining if flow rules match packets for the UDP protocol is more challenging since UDP is a connectionless protocol and will not send a reply message upon receiving a UDP packet. To evaluate the reception of a UDP packet, SDNMap transmits a probing packet to a port which is likely closed at the destination host, to trigger the destination to send an ICMP *port unreachable message (type 3 code 3)* as defined in RFC 792 [14]. To construct such a UDP probing message, SDNMap selects a random source and destination port between the range *35000-65000*. There is no guarantee that a randomly selected port in this range will be closed at the destination host, but it is likely since studies such as [26] show that 90% of the used port numbers are in lower ranges.

The reception of UDP probing packets can also be confirmed by sending probing packets with spoofed IP source addresses to the destination host and listening for ARP request packets. ARP requests will be transmitted from the destination host in case the received packet source IP address is not present in the local ARP cache of the destination host.

Based on the reply packets received by transmitting the probing packets, SDNMap can reconstruct the usage of specific source and destination port number combinations and protocol usage. As a result of this, SDNMap determines the values *PT* and *PO* to identify if their associated OpenFlow fields are used as a matching criteria in the flow rules forwarding packets to the scanned destination host.

If the scanning process determined that ICMP, TCP and UDP packets are delivered by the flow rule, SDNMap will reconstruct a single rule that matches packets for an IP header as specified by the OpenFlow protocol. Otherwise a separate rule for each delivered protocol is reconstructed.

3.5 Scanning step: Ingress port

Besides using header information of packets as a matching criteria, SDNMap infers if the field *match:in_port=SIP* is used as part of the matching criteria in a flow rule, but does not determine the actual port number *SIP*. A packet's ingress port number is determined by the physical or logical port to which a host is connected to an OpenFlow-enabled switch. Unless a host physically changes its connection to a different port, the ingress port number cannot be manipulated by a user. To determine if the ingress port, which can be classified as packet meta information, is used as a matching criteria in a flow rule, SDNMap performs a number of checks based on certain assumptions. A core assumption we make for determining this matching criteria is that flow rules connecting the SDNMap host to the probed host are constructed similar as the flow rules from the probed host to other nodes in the same sub-network.

As the first step to determine if the ingress port is used as a matching criteria, our scanner has to find two hosts in the sub-network which can be contacted by the SDNMap

host and are able to connect to each other. As an assumption we can start by selecting two random nodes that are reachable by the probing host as previously determined by the performed ARP scan.

After the selection of two hosts, *A* and *B*, SDNMap sends a probing packet with the source address of *A* to *B* as shown in Figure 3. Upon reception of the probing packet, *B* will lookup *A's* MAC address in it's local cache. If no cache entry is present, *B* will broadcast an ARP request that will also be received by SDNMap's host node. The reception of the ARP request indicates to SDNMap that the previously transmitted probing packet, as shown on the left in Figure 3, was received at B. Since SDNMap transmitted the probing packet with *A's* source addresses, but from a different ingress port than *A's*, it can be concluded that the ingress port is not checked as part of the flow rule matching criteria.

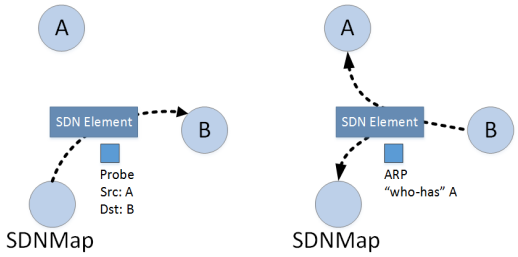

Figure 3: Initial steps to determine ingress port field

If the above procedure does not trigger the broadcast of an ARP request from *B*, SDNMap also considers the case when *A's* MAC address is already present in *B's* ARP cache. If this is true, *B* will not broadcast an ARP request upon receiving a probing message from *A*, which can be overheard by SDNMap to determine if the probe was received. In such a case, SDNMap will exploit the feature of validating an ARP cache entry with a unicast probe as discussed in RFC 1122 [12]. To trigger this procedure SDNMap will spoof *A's* ARP cache entry at *B* to point to the MAC address of SDNMap's host. This will appear to host *B* that *A* recently has changed its assigned MAC address, (e.g. if a different device got *A's* IP address assigned) and according to RFC 1122 a unicast ARP request will be sent to *A's* spoofed MAC address that is present in *B's* ARP cache to validate the cache entry. This unicast ARP validation request will be received by SDNMap's host since we previously set *A's* ARP cache entry at *B* to this MAC address. The reception of this unicast ARP validation request will indicate to SDNMap that the initial probing message that was sent to *B* on behalf of *A* was received and therefore the ingress port is not checked as part of the matching criteria.

If after executing the described procedure with multiple different host pairs *A* and *B*, no ARP request was received by SDNMap, our software concludes that none of the transmitted probing messages was received. Based on that, we can assume that OpenFlow field *match:in_port=SIP*, is part of the matching criteria of the evaluated flow rule to prevent the transmission of a probing packet on an incorrect switch ingress port. In this scanning step, SDNMap does not determine the actual value of the ingress port, since this information is only available inside the actual switch, but assigns the placeholder *#IN_PORT* to the ingress port field in the reconstructed flow rules.

3.6 Scanning step: Rewriting IP addresses

In this step we determine the fields *actions:set_IP_src=rIPs* and *actions:set_IP_dst=rIPd* are used as actions in a flow rule, and infers the assigned IP addresses *rIPs* and *rIPd*. To determine if an OpenFlow-enabled switch performs IP rewriting actions, SDNMap sends a UDP probing packet to a port which is very likely closed at the destination host. The source and destination port for the probing packet are randomly selected in the range *35000-65000*, as also discussed in Section 3.4. If the selected destination port is closed at the receiver, it will trigger the generation of an *ICMP - Destination Port Unreachable* message (type 3 code 3) that is sent back to the probing host.

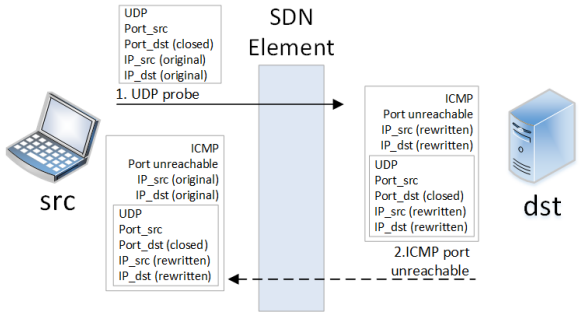

Figure 4: Steps to determine IP address rewriting

According to RFC 792 [14], an ICMP destination unreachable message will contain the first 64 bits of the original datagram. In this case, the original datagram is the probing packet SDNMap sent to the destination host. If the flow rule forwarding packets between the source and destination host also performed a rewriting action of the packets IP addresses, the encapsulated original datagram in the ICMP unreachable message will contain the IP addresses as received at the destination host. We show this procedure in Figure 4 where the encapsulated probing packet has different IP addresses than the ICMP destination unreachable message that is sent back to the probing host. The generating of an ICMP destination unreachable message will make the destination host reveal the received IP address fields of the probing packet, which enables us to conclude if an IP rewriting action was performed on the packet in the OpenFlow-enabled switch.

In OpenFlow [9], including version 1.5, actions for rewriting the header information of encapsulated datagrams are not provided. The leakage of rewritten IP addresses back to the original sender as explained, enables our SDNMap

scanner to determine if the received IP addresses at the destination are equal to those that were sent. If the IP addresses in the encapsulated datagram are different, SDNMap will conclude that an IP address rewriting action was performed by the flow rule. This enables our software to determine the values *rIPs* and *rIPd* and the usage of their associated OpenFlow fields *actions:set_IP_src=rIPs* for rewriting the IP source address and *actions:set_IP_dst= rIPd* for rewriting the IP destination address in a rule.

3.7 Scanning step: Forwarding action

An essential part of the actions performed in a flow rule is forwarding or dropping of packets. If a packet's header information is matching with the criteria defined in a rule's match fields, and optional actions such as address rewriting are performed, a forwarding action is executed. Such forwarding actions either send packets out of the appropriate port on a network element to reach its destination, or drop a packet in case of a security aware SDN application, such as a firewall. For sending a packet out of a specific port the OpenFlow action field *actions:output=#OUT_PORT* will be applied, or in case a packet is dropped *actions:DROP* will be used in the flow rule. SDNMap will reconstruct the action if a packet is forwarded or dropped, based on analyzing which probing packets were transmitted during the scanning steps discussed in Section 3.2, 3.3 and 3.4 received reply messages.

4 EVALUATION

In this section we evaluate our SDNMap scanner by applying it on a number of existing scenarios and applications to demonstrate that security issues arise when SDN flow rules can be reconstructed. We would like to point out that the discussed scenarios are real-world applications from different vendors which are implemented on existing SDN platforms and available online. Here, the composition of SDN flow rules is defined by the vendors of these applications.

We installed and replicated these scenarios in our test environments and strictly followed the supplied manuals to configure and run the applications in our SDN test networks.

The attack approaches we demonstrate are different from attacks on SDN discussed in the literature. While most current attack vectors on SDN target reactive deployments, to our best knowledge SDNMap is the first attack vector targeting reactive and proactive SDN deployments. Previously, attacks such as poisoning network topology or fabricating links [21], saturating the link between control- and data plane [29] or exploiting conflicting flow rules in firewall implementations [22], did not consider the approach of predicting the exact details of flow rules and use them as an attack advantage.

The presented scenarios show a subset of the applications we used to evaluate our techniques. We used both a software- and hardware-testbed to evaluate SDNMap.

4.1 Test environment

For testing and evaluating SDNMap we focus on SDN environments such as enterprise networks in organizations, local area or campus networks or networks in data centers and server farms. We tested its functionality on a testbed with software SDN switches (*OpenVSwitch* [10]) and on a testbed with hardware SDN switches (*Brocade ICX 6610* [1]). We evaluated SDNMap on three widely used open-source controllers, *POX* [11], *Floodlight* [6] and *OpenDayLight* [8].

As network endpoints, we use personal computers with Ubuntu 14.04 LTS 64bit. OpenFlow versions 1.0 and 1.3 were used for the deployment of flow rules.

4.2 Performance of rule reconstruction

To evaluate the rule reconstruction capability of SDNMap we tested it on a number of existing applications.

In the SDN network scenarios we replicated the reconstruction of flow rules from the SDNMap host to a destination host in the network takes 20 seconds on average. This scanning delay depends mostly on the performed scanning steps and defined timeout thresholds in SDNMap, and less on different network topologies. If a range of hosts and a list of protocol ports is scanned, the reconstruction process will take appropriately longer since several steps have to be performed multiple times.

To evaluate the accuracy of reconstructing flow rules, we denote the number of OpenFlow fields in a rule with F, the number of correctly determined flow rule fields as f and the number of incorrectly identified flow rule fields as m. We calculate the flow rule reconstruction accuracy α as shown in Equation 1.

$$\alpha = \frac{f - m}{F} \tag{1}$$

In Table 2 we show the accuracy SDNMap achieved in different application scenarios. In Sections 4.3 to 4.5 we present a detailed description of test-scenarios. In addition, we also evaluated SDNMap on a SDN-based firewall *(FW)*, a role-based access controlled SDN-network defined by Brocade [2] *(RBAC)* and a multi-hop proactive topology SDN network *(MHPA)* as listed in Table 2, but have to omit a detailed description of these scenarios due to space limitations. On average, SDNMap achieves an accuracy of over 96% on all evaluated scenarios.

Table 2: SDNMap flow rule prediction accuracy

	ACL Sec. 4.3	**LBaaS** Sec. 4.4	**MTD** Sec. 4.5	**FW** -	**RBAC** -	**MHPA** -
α	1	1	0.83	0.93	1	1

The reason if SDNMap is not able to correctly reconstruct a rule to 100%, is that some scanning steps could not resolve all required information for the reconstruction of fields, such as not finding appropriate neighbor nodes to determine the ingress port field as discussed in Section 3.5.

To scan a destination host with the introduced rule- reconstruction techniques discussed in Sections 3.2 to 3.7, SDNMap needs to transmit $14 + 2p^2$ packets, where p is the user-specified list of port numbers evaluated during the scanning process. SDNMap will evaluate all possible source and destination combinations of the supplied port numbers for TCP and UDP packets, therefore $2p^2$ probes are required.

In the following application scenarios we will present how the ability of flow rule reconstruction can be used by adversaries as a reconnaissance tool to plan and execute further attack maneuvers.

4.3 Floodlight Access Control List

The open-source Floodlight controller includes a number of applications, such as the implementation of an Access Control List (ACL) which deploys rules on an OpenFlow enabled switch to control network access. As explained on the Floodlight webpage [6], the ACL implementation enables users to allow or deny access to hosts in a network. For the deployment of flow rules in this scenario, we follow the configuration examples listed on the Floodlight webpage [6].

To demonstrate SDNMap's functionality in such a scenario we created a small network of six hosts connected by an SDN switch. By following Floodlight's ACL manual we are using its REST interface to proactively deploy rules for denying access to the host at *10.0.0.2* by host *10.0.0.1* by executing the following commands:

"src-ip":"10.0.0.1/32","dst-ip":"10.0.0.2/32","action":"deny"
"src-ip":"10.0.0.2/32","dst-ip":"10.0.0.1/32","action":"deny"

We used SDNMap to perform a scan of the network *10.0.0.0/24* from the node with address *10.0.0.1*. The scanning results of SDNMap labeled host *10.0.0.2* as being online, reported that a learning approach is used on the paths from *10.0.0.1* to *10.0.0.3*, *.4*, *.5* and *.6*, and reconstructed the following rules for the path *10.0.0.1* to *10.0.0.2*:

match=type:ip,nw_src:10.0.0.1,nw_dst:10.0.0.2 actions=drop
match=type:ip,nw_src:10.0.0.2,nw_dst:10.0.0.1 actions=drop

The reconstructed flow rules by SDNMap are an exact match with the previously deployed access control rules. An attacker using SDNMap in such a scenario is now able to determine how packets have to be crafted to be successfully delivered to the host at *10.0.0.2*. For example, sending a packet with the correct destination address but a source IP address different than *10.0.0.1*, will be delivered to host *10.0.0.2* since the Floodlight controller is forwarding packets not matching ACL rules with the default learning approach.

Besides blocking access from host *10.0.0.1* to host *10.0.0.2*, the path to host *10.0.0.2* still follows the default learning approach implemented by the Floodlight controller. In our testbed we created a scenario where host *10.0.0.2* runs a webserver hosting files which are not supposed to be accessed by host *10.0.0.1*. As discussed, we used Floodlight's ACL implementation to deny access from host *10.0.0.1* to host *10.0.0.2*. By using SDNMap we retrieved the flow rule composition as demonstrated. Based on SDNMap's result we sent a *HTTP GET* request packet with a spoofed IP source address from *10.0.0.1* to *10.0.0.2*. Since traffic not matching the access control rules is delivered based on the default learning approach, the packets from *10.0.0.1* with a spoofed IP source address were forwarded to *10.0.0.2* and the HTTP response traffic was delivered back to *10.0.0.1*. Such a scenario can be imagined to be deployed for denying access to a company's Intranet for guest users who are temporarily connecting to the company's network. With SDNMap malicious users are able to reconstruct the deployed access control policy and bypass it with appropriately crafted packets as demonstrated.

4.4 Load Balancing as a Service

The Floodlight SDN controller provides a load balancer module which is defined by the OpenStack Quantum LBaaS (Load-Balancing-as-a-Service) API. The LB- aaS API provides a framework to virtualize networking resources and balances client requests between the available physical resources. We followed the setup instructions for the LBaaS module as discussed on the Floodlight webpage [5].

In such a setup, a client requests access to a resource which is mapped onto a virtual network topology. Such a request is forwarded to a physical network resource by translating packet information, such as IP addresses, accordingly. This functionality is implemented with the use of OpenFlow rules that are deployed on the SDN network elements.

The provided LBaaS API allows network administrators and cloud operators to configure the load balancer by defining a list of virtual IP addresses, a list of physical network endpoints and resource pools to map the handling of virtual resources to physical resources. Following the configuration manual on the Floodlight webpage [5], we specified the virtual network endpoint *10.0.0.150* which is assigned to a resource pool that is handled by the physical network endpoints *10.0.0.7* and *10.0.0.8*.

The generated flow rules to implement this load balancing function are deployed over multiple SDN network elements and forward traffic to their assigned physical network endpoint based on their IP source address. Traffic to the virtual endpoint *10.0.0.150* from the client endpoint *10.0.0.1* is handled by the following flow rules which are automatically generated and deployed by the controller:

nw_src=10.0.0.1 actions=mod_nw_dst:10.0.0.7,output:#port
nw_dst=10.0.0.1 actions=mod_nw_src:10.0.0.150,output:#port

The deployed flow rules show that traffic originating from node *10.0.0.1* is forwarded based on its IP source address, and the destination address is translated to *10.0.0.7* to be handled by the physical network endpoint at this address. Return traffic to *10.0.0.1*, as defined by the second rule, has its IP source address rewritten to *10.0.0.150*, so that it appears to the client that it is communicating with *10.0.0.150* rather

than *10.0.0.7*.

Using our SDNMap tool to scan host *10.0.0.150*, we are able to reconstruct the following rule which reveals the load balancing functionality:

match=type:nw_src:10.0.0.1,nw_dst:10.0.0.150
actions=mod_nw_dst:10.0.0.7,output:#OUT_PORT

The reconstructed rule states that traffic to *10.0.0.150* is forwarded based on its source and destination address and shows that the IP destination address is rewritten to *10.0.0.7*. This reveals to a user that address *10.0.0.150* is actually handled by a server at *10.0.0.7*. A malicious user who scans a range of virtual IP addresses is able to reconstruct the implemented load balancing policy, and launch a targeted denial of service attack on specific virtual IP addresses which all map to the same server, to overwhelm a specific physical network resource.

The ability to reconstruct load balancing mechanism which are implemented based on SDN for cloud platforms, such as OpenStack, represent a severe security leakage since adversaries can use this knowledge for launching attacks to target specific endpoints in such systems and disrupt the provided service for cloud customers.

4.5 Moving Target Defense

It has been proposed in recent publications [23, 24], to use SDN rule functionalities as a Moving Target Defense technique, by randomizing the address space of a network with high frequency. This is done by using the packet header field modification functionalities provided by OpenFlow. Using such techniques can defend against network adversaries, such as computer worms, performing network scans and trying to map the address space of a network to identify targets and perform further attacks.

We implemented the *OpenFlow Random Host Mutation (OF-RHM)* defense mechanisms proposed in [23, 24] in the POX controller framework and randomize the IP address space with high frequency (every 5 seconds). SDNMap is able to successfully reconstruct the flow rules used for the address space randomization. On average SDNMap required ~4 seconds to report during the scanning process that the IP addresses of a packet are rewritten while it passed through an SDN switch and was forwarded to its destination. To show an example, during the scanning process in our test environment of host *10.0.0.97*, SDNMap reported the rewriting of packet header information on the path:

*S*ending UDP packet to port 36028 at 10.0.0.97 / 00:00:00:00:00:05
Received ICMP Port Unreachable message
IP src 10.0.0.1 is rewritten to 10.0.0.97
IP dst 10.0.0.97 is rewritten to 10.0.0.5

This reflects precisely the functionality that was implemented by the OF-RHM controller to hide the true IP address of network endpoint *10.0.0.5* in the example above. An adversary using SDNMap can perform such a scan multiple times, determine that the network implements a moving target defense technique and collect the real number and IP addresses of hosts in such a network, which is supposed to be only known by the network operator or administrator.

5 DEFENDING RULE RECONSTRUCTION

While the focus of this paper is on demonstrating the possibilities and impacts of SDN flow rule reconstruction techniques, we want to discuss defense techniques to prevent this new attack vector. We summarize existing, SDN specific defense techniques and discuss that such techniques are ineffective against multiple rule reconstruction techniques presented in this paper.

We also present an overview of methods to defend SDNMap, but leave a detailed analysis of such techniques for future work.

5.1 Current defense techniques

In this section we discuss recently published defense techniques against adversarial actions in SDN. The proposed defense approach Sphinx [20] evaluates *Flow_mod, Stats_reply, Packet_in* and *Features_reply* messages, which are exchanged between control and data plane for adversarial behavior. While the proposed defense approach is novel and effective against certain attack actions in SDN, it assumes a static network environment by depending on fixed IP/MAC and MAC/Port bindings. In dynamic environments with changing network conditions, which are pertinent for the usage of SDN, such a configuration would be hard to maintain and might not be feasible as discussed in [21].

The authors of [20] state that Sphinx does not raise alarms when new flow behavior is discovered, but raises alarms when changes in existing flow behavior is found. The approach we use in SDNMap does not aim to change existing flow behavior, or manipulate the topology, but maps the details of existing flow behavior.

To prevent Sphinx from triggering false alarms, the authors define $1/\tau < \sum_{n+1}/\sum_{avg} < \tau$, where τ is manually configured, so that $\tau = 1.045$ by default. $\sum$ is defined as the similarity-index and calculated as the moving average of the difference in byte-level statistics, where $\sum_{avg}$ is the average similarity index of a flow, and $\sum_{n+1}$ is the index for the next switch in a flow. The authors state that Sphinx will not raise alarms if $1/\tau < \sum_{n+1}/\sum_{avg} < \tau$ holds true.

As stated in Section 4.2, SDNMap transmits $14+2p^2$ packets to reconstruct the flow rules to a destination host. For a set of ports to scan, e.g. $p = \{22, 80\}$, SDNMap will transmit 22 packets. With the average scanning time of 20s, SDNMap would transmit ~1 (22/20) probing packets per second on average. In the evaluations presented in [20], scenarios with flows of 100-10000 packets per second are assumed. Using the lower bound of such scenarios, $0.96 < (1 + 100)/100 < 1.045$,

holds true and SDNMap would stay "under the radar" of the detection system.

Scanning multiple destination hosts is executed sequentially in SDNMap and will not exceed the detection threshold of defense systems such as Sphinx, as shown above. The authors of [20] state that Sphinx does not work for proactive SDN deployments. SDNMap does not attack the controller platform in SDN deployments and therefore works equivalently on reactive and proactive deployments. Based on the presented analysis it can be assumed that SDNMap's activity would be insensitive to systems such as Sphinx.

5.2 Rule reconstruction defense discussion

In Section 4 we demonstrate how a number of real-world applications can be attacked given an adversary is able to determine the details of flow rules. The defense mechanism we discuss in this section represents a list of potential ideas which we are planning to further evaluate in our future work.

Handling of ARP traffic by the controller:
As discussed in papers and by networking companies [3, 20, 21], ARP poisoning is an existing problem, especially in modern networking concepts such as SDN, and static network configurations are impractical defense approaches. In SDN, rules forwarding broadcasted ARP requests to the controller can be pro-actively installed on network elements. Since the controller maintains a global network view, ARP requests can be handled by the controller which is able to reply on behalf of the actual hosts. Such a setup will prevent the leakage of information caused by broadcasting ARP requests and avoid ARP cache poisoning on network endpoints. **Information leakage through nested packets:**
We demonstrate in Section 3.6 how SDNMap is able to infer address rewriting actions by triggering ICMP error reply messages. To prevent such information leakage, an SDN controller should adjust nested packets according to the implemented network policy to prevent adversaries from revealing actions, such as the adjustment of header fields.

Definition of *best practice* for OpenFlow rules:
In the existing SDN applications we analyzed in our evaluations, we observe that no common standard is used to define which fields to use for the construction of OpenFlow rules. Identifying packets only based on destination addresses or without considering the switch ingress port, will provide attackers with an increased flexibility for generating adversarial traffic by spoofing packet information that is not used in the matching criteria. Even if adversaries are able to reconstruct flow rules, defining best-practice for their construction will reduce the ability for attackers to transmit crafted traffic.

6 RELATED WORK

In [19] and [28] the authors discuss the possibilities of fingerprinting SDN networks. Their work is focused on determining if a network is SDN based by analyzing the timing distribution of active probing messages as well as passive observations of the network traffic. Both papers demonstrate how the timing distribution of packets on the network can reveal an SDN network and retrieve certain details about the network topology, for example the number of SDN switches visited on a path in the network. Neither [19] nor [28] considers analyzing the deeper functionality of SDN networks defined by the detailed composition of flow rules we show how to reconstruct in this paper.

In [29] the framework *Avant-Guard* is introduced which addresses the challenges of control plane saturation attacks in SDN architectures where an attacker aims to overflow the bottleneck between SDN network elements (switches) and the SDN controller. Another challenge addressed in this paper is to expedite the detection of, and the responses to certain changes in a network's flow dynamics. The *Avant-Guard* framework is an extension of the data plane that enables scalable and resilient security aware SDN architectures.

The authors of [31] present *OrchSec* an orchestration based architecture for the development of SDN based security applications. The development process of *OrchSec* is based on analyzing SDN characteristics, such as a centralized management or network visibility, for the development of security aware network applications. The work points out the deficiencies of SDN for security applications and proposes several architectural requirements to adapt the architecture of SDN for security use cases.

7 CONCLUSION

In this paper we demonstrate that flow rules in SDN networks can be predicted and reconstructed in detail by a user connected to the network. Using our techniques, an adversary is able to reveal implementation details of network policies based on OpenFlow rules, and can use this knowledge to exploit the flow rule composition for further malicious actions. We discuss real-world SDN application scenarios, and point out that the predictability of flow rules can open severe security leaks if exploited by attackers. To prevent this and make SDN-based network systems more secure, we briefly discuss defense approaches, such as defining best practices for the construction of OpenFlow rules.

ACKNOWLEDGMENT

The effort described in this article was sponsored by the U.S. Army Research Laboratory Cyber Security Collaborative Research Alliance under Cooperative Agreement W911NF-13-2-0045. The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation hereon.

REFERENCES

[1] Brocade icx 6610. http://bit.ly/2d3TSBH. Accessed: 2016-04-20.
[2] Brocade openflow developer guide. http://bit.ly/2cXwnfh. Accessed: 2016-04-21.
[3] Cisco configuring port security. http://bit.ly/ZfVjpD.
[4] Firewalk firewall scanner. https://packetstormsecurity.com/UNIX/audit/firewalk/.
[5] Floodlight lbaas. http://bit.ly/2d6gKUY. Accessed: 2016-06-12.
[6] Floodlight sdn controller. http://www.projectfloodlight.org/floodlight/. Accessed: 2016-04-20.
[7] Is an sdn switch a new form of a firewall? http://bit.ly/1xDhVAo.
[8] Opendaylight sdn controller. https://www.opendaylight.org/. Accessed: 2016-04-20.
[9] Openflow. https://www.opennetworking.org/sdn-resources/openflow.
[10] Openvswitch. http://openvswitch.org/.
[11] Pox sdn controller. http://stanford.io/2ctlCEf. Accessed: 2016-04-20.
[12] Rfc1122 requirements for internet hosts. tools.ietf.org/html/rfc1122.
[13] Rfc5227 ipv4 address conflict detection. tools.ietf.org/html/rfc5227.
[14] Rfc792 internet control message protocol. tools.ietf.org/html/rfc792.
[15] Rfc793 transmission control protocol. tools.ietf.org/html/rfc793.
[16] Rfc826 an ethernet address resolution protocol. tools.ietf.org/html/rfc826.
[17] Sdnmap repository. https://github.com/SDNMap/sdnmap.
[18] Ali, M. Q., Al-Shaer, E., and Samak, T. Firewall policy reconnaissance: Techniques and analysis. *IEEE Transactions on Information Forensics and Security* (2014).
[19] Bifulco, R., Cui, H., Karame, G. O., and Klaedtke, F. Fingerprinting software-defined networks. In *2015 IEEE International Conference on Network Protocols (ICNP)*.
[20] Dhawan, M., Poddar, R., Mahajan, K., and Mann, V. Sphinx: Detecting security attacks in software-defined networks. In *NDSS* (2015).
[21] Hong, S., Xu, L., Wang, H., and Gu, G. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *NDSS* (2015).
[22] Hu, H., Han, W., Ahn, G.-J., and Zhao, Z. Flowguard: building robust firewalls for software-defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking* (2014), ACM.
[23] Jafarian, J. H., Al-Shaer, E., and Duan, Q. Adversary-aware ip address randomization for proactive agility against sophisticated attackers. In *Computer Communications (INFOCOM), 2015 IEEE Conference on*.
[24] Jafarian, J. H., Al-Shaer, E., and Duan, Q. Openflow random host mutation: transparent moving target defense using software defined networking. In *Proceedings of the first workshop on Hot topics in software defined networks* (2012).
[25] Khakpour, A. R., Hulst, J. W., Ge, Z., Liu, A. X., Pei, D., and Wang, J. Firewall fingerprinting. In *INFOCOM, 2012 Proceedings IEEE*.
[26] Lee, D., Carpenter, B. E., and Brownlee, N. Observations of udp to tcp ratio and port numbers. In *IEEE Internet Monitoring and Protection (ICIMP), 2010*.
[27] Samak, T., El-Atawy, A., and Al-Shaer, E. Firecracker: A framework for inferring firewall policies using smart probing. In *2007 IEEE International Conference on Network Protocols*.
[28] Shin, S., and Gu, G. Attacking software-defined networks: A first feasibility study. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking* (2013), ACM.
[29] Shin, S., Yegneswaran, V., Porras, P., and Gu, G. Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*.
[30] Wool, A. Architecting the lumeta firewall analyzer. In *USENIX Security Symposium* (2001).
[31] Zaalouk, A., Khondoker, R., Marx, R., and Bayarou, K. Orchsec: An orchestrator-based architecture for enhancing network-security using network monitoring and sdn control functions. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*.