When Match Fields Do Not Need to Match: Buffered Packet Hijacking in SDN

Jiahao Cao[*†‡§], Renjie Xie[*‡§], Kun Sun[†], Qi Li[‡§], Guofei Gu[¶], and Mingwei Xu[*‡§]
[*]Department of Computer Science and Technology, Tsinghua University
[†]Department of Information Sciences and Technology, George Mason University
[‡]Institute for Network Sciences and Cyberspace, Tsinghua University
[§]Beijing National Research Center for Information Science and Technology, Tsinghua University
[¶]SUCCESS LAB, Texas A&M University
{caojh15, xrj16}@mails.tsinghua.edu.cn ksun3@gmu.edu {qli01, xumw}@tsinghua.edu.cn guofei@cse.tamu.edu

***Abstract*—Software-Defined Networking (SDN) greatly meets the need in industry for programmable, agile, and dynamic networks by deploying diversified SDN applications on a centralized controller. However, SDN application ecosystem inevitably introduces new security threats since compromised or malicious applications can significantly disrupt network operations. Thus, a number of effective security enhancement systems have been developed to defend against potential attacks from SDN applications. In this paper, we identify a new vulnerability on flow rule installation in SDN, namely, *buffered packet hijacking*, which can be exploited by malicious applications to launch effective attacks bypassing all existing defense systems. The root cause of this vulnerability lies in that SDN systems do not check the inconsistency between buffer IDs and match fields when an application attempts to install flow rules. Thus, a malicious application can manipulate buffer IDs to hijack buffered packets even though they do not match any installed flow rules. We design effective attacks exploiting this vulnerability to disrupt all three SDN layers, i.e., application layer, data plane layer, and control layer. First, by modifying buffered packets and resending them to controllers, a malicious application can poison other applications. Second, by manipulating forwarding behaviors of buffered packets, a malicious application can not only disrupt TCP connections of flows but also make flows bypass network security policies. Third, by copying massive buffered packets to controllers, a malicious application can saturate the bandwidth of SDN control channels and their computing resources. We demonstrate the feasibility and effectiveness of these attacks with both theoretical analysis and experiments in a real SDN testbed. Finally, we develop a lightweight defense system that can be readily deployed in existing SDN controllers as a patch.**

I. Introduction

Software-Defined Networking (SDN) has emerged as a flexible network paradigm. It is being increasingly deployed in enterprise data centers, cloud networks, and virtualized environments [1], [2]. The popularity of SDN lies on its programmability, agility, and dynamic network control, which benefits from the separation of control and data planes. SDN allows a logically centralized controller in the control plane to control all SDN switches in the data plane. By deploying diversified applications on the controller, various network functionalities can be easily implemented in SDN, such as load balancing [3], traffic engineering [4], and network security forensics [5]. Network functionalities that are formerly implemented by proprietary software or complex middleboxes now can be enabled through the use and update of SDN applications from open source developer communities or third-party application stores [1], [6], [7].

Though SDN applications extend the capacities of controllers and bring huge benefits, the burgeoning application ecosystem introduces new security threats. SDN applications may have buggy or malicious code that can be exploited by attackers to disrupt network operations. A recent study [8] shows that malicious applications are probably the most severe threats to SDN. A number of attacks [1], [9], [10], [11], [7] launched by malicious applications have been identified, such as exploiting shared data objects in controllers to poison applications [1], manipulating flow rules to bypass network security policies [9], [10], and abusing permissions to crash SDN controllers [11].

To prevent malicious applications from disrupting SDN systems, a number of effective defense systems have been proposed. Permission control systems can effectively limit excessive privileges of applications [12], [7], [13], [14]. Data provenance systems can prevent cross-app poisoning by tracking shared data objects in controllers and checking information flow control (IFC) [1]. Rule conflict detection systems can deter attackers from bypassing network security policies, which is introduced by rule manipulation from malicious applications [10], [9], [15]. Sandbox systems can protect SDN controllers against malicious operations performed by isolated applications [16]. All these defense mechanisms significantly improve the security of SDN and raise the bar for a malicious application to launch attacks.

In this paper, we uncover a new vulnerability on flow rule installation in SDN, namely, *buffered packet hijacking*. It allows a malicious application to hijack buffered packets to launch a number of attacks bypassing existing defense systems. The vulnerability is due to the lack of consistency check between buffer IDs and match fields when installing flow rules. Typically, an SDN application sends a `FLOW_MOD` message to switches when a new flow matches no flow rules. The message not only contains *match fields* and *actions* to

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.23040
www.ndss-symposium.org

create flow rules in switches for the new flow, but also contains a *buffer ID* to release a previously buffered packet of the new flow. However, we find that the buffered packet specified by the buffer ID can be directly forwarded according to the actions, no matter if it matches installed flow rules. Thus, a malicious application can pretend to update flow rules under its responsibility while it manipulates buffer IDs in `FLOW_MOD` messages to stealthily control the forwarding behaviors of buffered packets for any new flows. Though buffered packets do not match the flow rules installed by the malicious application, they will be processed according to the actions of the flow rules. As the malicious flow rules do not conflict with other flow rules installed by benign applications, existing defense systems cannot detect or prevent these hijacking operations of a malicious application [10], [9], [15].

We note that successfully hijacking buffered packets requires the malicious application to send a `FLOW_MOD` message before a benign application sends its message to release buffered packets. This depends on processing chains that define the orders for different SDN applications to process network events. To analyze the probability of successfully hijacking a buffered packet, we build a model of processing chains that is independent of controllers. We derive the formal representation of the hijacking probability in two typical scenarios, i.e., intra-chain hijacking and inter-chain hijacking.

Based on the vulnerability, we discover four attacks that can disrupt all three layers of SDN, i.e., application layer, data plane layer, and control layer. All these new attacks can successfully evade all existing defense systems. First, we discover a cross-app poisoning attack that targets at the application layer. By modifying a buffered packet and resending it to controllers, a malicious application can poison other applications that learn information from the headers of buffered packets. Second, we uncover two attacks targeting at the data plane layer, namely, network security policy bypass attack and TCP three-way handshake disruption attack. A malicious application can easily launch these two attacks by modifying forwarding behaviors of the buffered packets. Finally, we discover a control traffic amplification attack that targets at attacking the control layer by copying massive buffered packets to controllers. It quickly increases control traffic of SDN and thus consumes both bandwidth of control channels and computing resources of controllers.

We conduct experiments in a real SDN testbed consisting of commercial hardware SDN switches and open source SDN controllers to demonstrate the feasibility and effectiveness of the identified attacks. The experimental results show that the hijacking probability can exceed 70% in most cases of real processing chains, and the largest hijacking probability can reach 100%. Our experimental results are consistent with the theoretical results. Moreover, we demonstrate that a malicious application can successfully launch a number of effective attacks by hijacking buffered packets. In the application layer, a malicious application can poison the learning switch application to falsely learn the mappings between MAC addresses of hosts and switch ports. In the data plane layer, malicious data flows can successfully bypass network security policies due to the modification of forwarding behaviors of buffered packets. Furthermore, by disrupting TCP three-way handshake, a malicious application can significantly delay the connection completion time (CCT) for TCP flows, which is 100 times higher than that in normal cases. In the control layer, the control channel is quickly saturated under control traffic flooding attacks, leaving most flows not served in SDN.

To prevent attacks from hijacking buffered packets, we develop a lightweight countermeasure named *ConCheck*. It is transparent to SDN applications and can be easily deployed on SDN controllers as a patch. ConCheck intercepts API calls of reading `PACKET_IN` messages in order to build mappings between buffered packets and buffer IDs. With the knowledge of the mappings, it can check if there is any inconsistency between buffered packets and the match fields in API calls of generating `FLOW_MOD` messages. ConCheck blocks the API call that has any inconsistency to prevent hijacking buffered packets. We implement ConCheck on the Floodlight controller, and the experiments show it only introduces a small overhead.

In summary, our paper makes the following contributions:

- We identify a vulnerability on flow rule installation in SDN, which allows a malicious application to hijack buffered packets and evade existing defense systems.
- We develop four effective attacks that exploit the identified vulnerability to attack all three SDN layers.
- We conduct experiments in a real SDN testbed to demonstrate the hijacking probability and effectiveness of the identified attacks.
- We design and implement a lightweight countermeasure named ConCheck to prevent hijacking buffered packets.

II. Background

Multiple SDN applications concurrently run on controllers to enable diversified network functionalities. They interact with core services in controllers to obtain abstracted network state and enforce commands to control SDN switches. An SDN switch can leverage the `PACKET_IN` mechanism to actively report to SDN controllers that a new flow comes. We briefly introduce the mechanism here. When the first packet of a new flow matches no flow rules in a switch, it is automatically assigned with a buffer ID by the switch and then temporarily buffered in the switch, as shown in Figure 1. Meanwhile, a `PACKET_IN` message is sent to the controller and then is dispatched to SDN applications. Any application with the `PACKET_IN` permission can obtain the packet headers and the buffer ID contained in the `PACKET_IN` message.

As the `PACKET_IN` message contains useful information of a flow, e.g., source and destination IP addresses, most applications analyze the message to make network decisions. They have a certain order on processing the `PACKET_IN` message from a switch due to their interdependence. Therefore, different applications make up multiple *processing chains* to process `PACKET_IN` messages, as shown in Figure 1. The first application in each processing chain simultaneously receives a copy of a `PACKET_IN` message dispatched by the `PACKET_IN` notifier. However, applications in the same processing chain process the `PACKET_IN` message and make network decisions in turn. If an application is not interested in a particular type of a `PACKET_IN` message, it simply passes the message to the next application in the processing chain. Otherwise, it makes network decisions by interacting with core services on controllers. After processing the `PACKET_IN`

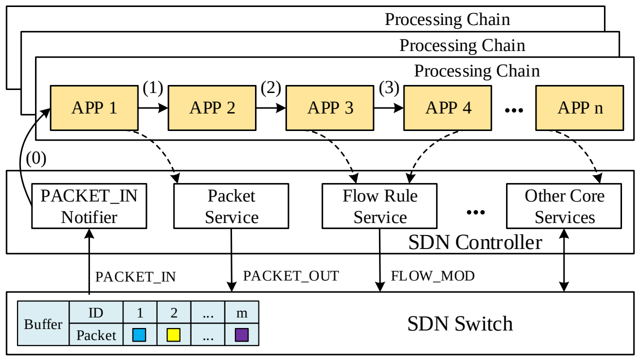


Fig. 1: Processing Chains in SDN.

message, it usually passes the message with possible metadata to the following application.

We use a real processing chain in the Floodlight [17] controller to show how applications process network events and enforce network policies. Considering the processing chain in Figure 1, we assume that APP 1, APP 2, APP 3, and APP 4 are *Topology Manager*, *Device Manager*, *Load Balancer*, and *Forwarding*, respectively. When the first application, i.e., Topology Manager, receives a `PACKET_IN` message (Step 0), it checks the type of the message. If the message contains an LLDP packet that is used for discovering links, it updates network topology and calls *Packet Service* to send new LLDP packets to switches for future topology discovery. Otherwise, it passes the message to the second application (Step 1), i.e., Device Manager. Device Manager learns attachment points of hosts by analyzing the `PACKET_IN` message and the network topology provided by the first application. After that, the message is passed to the third application (Step 2), i.e., Load Balancer, which checks the packet headers in the `PACKET_IN` message. If the new flow belongs to predefined important flows, Load Balancer chooses an optimal backend server from a server pool to serve the new flow. It calls *Flow Rule Service* to install crafted flow rules with `FLOW_MOD` messages. The new flow is then forwarded to the chosen server according to a particular routing path. Otherwise, it passes the `PACKET_IN` message to the following application (Step 3), i.e., Forwarding. Forwarding analyzes the message and calls Flow Rule Service to install flow rules with `FLOW_MOD` messages. The new flow is then forwarded to its destination with the shortest path.

III. Buffered Packet Hijacking

In this section, we first clarify our threat model and then present the buffered packet hijacking vulnerability in SDN.

A. Threat Model

We assume that the SDN controller, SDN switches, and control channels are trusted and well protected from attackers. However, since SDN applications installed on the controller may come from untrusted third-party SDN APP Store [6], they are untrusted and may be malicious. Previous studies have shown that malicious applications may be installed on controllers in many ways [1], [11], [18], [19], [7], e.g., exploiting particular vulnerabilities of controllers [19], repackaging and redistributing applications by phishing [18], and submitting malicious or buggy applications to a controller's repository that will be incorporated into commercial controllers [1]. Though network and security practitioners have made great efforts to verify the security of applications before deploying them by checking their source code, it is difficult to fully understand the behavior of compiled applications without getting access to the source code. For example, the ONOS controller [20] can load compiled applications. Therefore, SDN controllers may have potential malicious applications running on them.

TABLE I: The ratio of applications with the permission on listening `PACKET_IN` messages and installing flow rules.

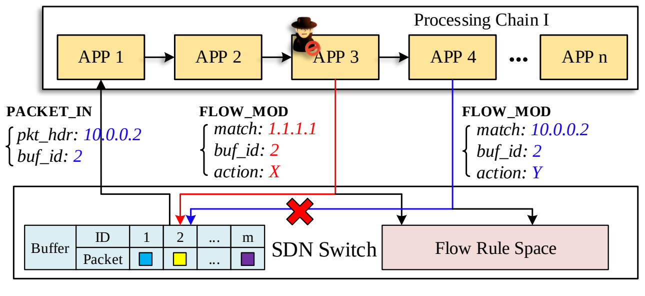
Controller	Total APPs	APPs with the Permission	Ratio
`OpenDaylight Neon`[†]	13	6	46.2%
`ONOS v2.1.0-rc1`	97	23	23.7%
`Floodlight v1.2`	29	12	41.4%
`RYU v4.31`	28	19	67.9%
`POX eel version`	18	11	61.1%

[†] Only counting the applications implemented with *openflowplugin*.

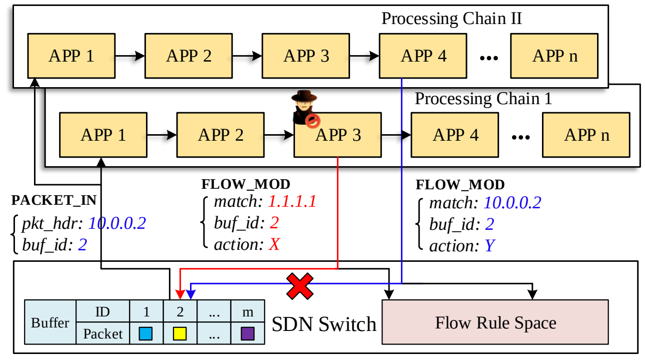
We assume that a malicious application is running on the controller. The application has the permission on listening `PACKET_IN` messages and installing flow rules, which is one basic requirement for many applications to run on the controller. We perform a study, and Table I shows the number and ratio of bundled applications with that permission on different controllers. The attacker aims to leverage the malicious application to disrupt all three SDN layers, i.e., application layer, control layer, and data plane layer.

We argue that a malicious application with that permission cannot directly install malicious flow rules to manipulate or drop packets in the SDN data plane. Otherwise, malicious rules will introduce conflicts with benign rules enforced by other SDN applications. Consequently, a number of effective defense systems, such as FortNOX[9], SE-FloodLight[10], VefiFlow[15], and SDNShield[7], can detect and prevent such permission abuse by checking rule conflicts between different applications. For example, a malicious application may install a flow rule with a high priority to drop all packets matching the 10.0.0.1 IP address, but the defense systems can detect a conflict when a benign routing application generates another flow rule that matches and forwards the packets with the same IP. Instead, our attacks install mismatched flow rules with crafted buffer IDs to hijack packets, which introduces no rule conflicts with other applications and thus bypasses existing defense systems.

Moreover, compared to the threat model in CAP attacks [1] where a malicious application manipulates shared data objects in controllers to poison other applications, our threat model requires a malicious application to have the flow rule installation permission. However, our attacks allow a malicious application to poison other applications even under the presence of the defense named ProvSDN [1] that can prevent CAP attacks. Furthermore, our attacks can disrupt not only the application layer but also the control layer and the data plane layer. Particularly, our attacks can bypass all the previous defense systems [12], [1], [10], [9], [15], [7], [16], [21] that prevent various attacks from disrupting different SDN layers. We detail our attacks and the mechanisms of bypassing existing defense systems in Section IV.



(a) Intra-chain Hijacking



(b) Inter-chain Hijacking

Fig. 2: Buffered Packet Hijacking in Processing Chains.

B. Buffered Packet Hijacking Vulnerability

A malicious application can hijack buffered packets to launch effective attacks that significantly disrupt all layers of SDN and bypass existing defense systems. Figure 2 shows the main idea of buffered packet hijacking in processing chains. For simplicity, we do not show the core SDN services of controllers in the figure. There are two types of buffered packet hijacking: intra-chain and inter-chain.

Intra-chain Hijacking. Figure 2a shows the intra-chain hijacking. A `PACKET_IN` message is dispatched to a processing chain since a new flow arrives at an SDN switch. As we mentioned before, the `PACKET_IN` message contains the headers and buffer ID of the first packet in the new flow. In the figure, the IP destination address of the packet headers is 10.0.0.2 and the buffer ID is 2. Normally, APP 4 is responsible for routing this flow. When the `PACKET_IN` message is transferred to the application, it installs a flow rule in the switch with a `FLOW_MOD` message. The message contains a match field specified as 10.0.0.2, a buffer ID specified as 2, and an action specified as Y. Thus, the new flow matches the installed flow rule and is processed according to the action Y. At the same time, the buffered packet with the buffer ID of 2 is released and processed according to the action.

Now suppose APP 3 is a malicious application that aims to disrupt SDN by hijacking buffered packets. Before the `PACKET_IN` message is transferred to APP 4, the malicious application can pretend to add or update flow rules for which it is responsible. It specifies the match field as 1.1.1.1, the buffer ID as 2, and the action as X in the `FLOW_MOD` message. Though the match field (1.1.1.1) and the header (10.0.0.2) of the packet with the buffer ID of 2 do not match, the buffered packet is still be released and processed according to the action X once the switch receives the `FLOW_MOD` message. Therefore, the buffered packet is processed by the malicious application before APP 4 installs flow rules to process it. The malicious application can hijack buffered packets of any new flow, as long as it is in the front of the application that is responsible for the new flow in the processing chain.

Inter-chain Hijacking. Figure 2b shows the inter-chain hijacking. Each processing chain simultaneously receives a copy of `PACKET_IN` message once a new flow arrives. Thus, a malicious application can hijack a buffered packet for which an application in another processing chain is responsible, using a similar method in Figure 2a, i.e., installing flow rules with a `FLOW_MOD` message that specifies the buffer ID of the packet. Different from the intra-chain hijacking, successfully hijacking buffered packets does not require that the malicious application is in the front of the benign application. In Figure 2b, it is possible that APP 3 in Processing Chain I hijacks buffered packets that should be processed by APP 2 in Processing Chain II since different applications consume different time to process and transfer a `PACKET_IN` message to another application. APP 3 in Processing Chain I may receive the `PACKET_IN` message before APP 2 in Processing Chain II. In some cases, APP 3 in Processing Chain I may fail to hijack buffered packets since another benign application has already processed the packets. Successfully hijacking buffered packets depends on the positions of the malicious application and the time for applications to process `PACKET_IN` messages. We will give a comprehensive theoretical analysis in Section V and experimental analysis in Section VI-B.

TABLE II: SDN controllers with buffered packet hijacking vulnerability.

Controller	OpenDaylight	ONOS	Floodlight	RYU	POX
Latest Version	Neon	v2.1.0-rc1	v1.2	v4.31	eel
Vulnerable	√	√	√	√	√

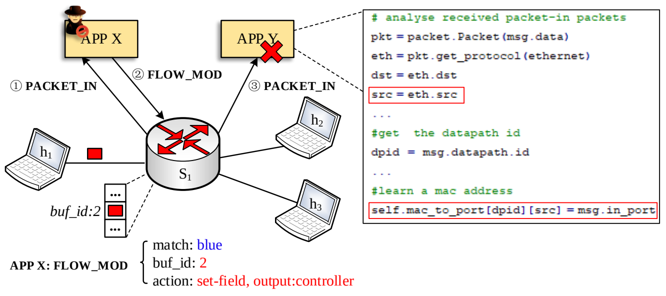
We have tested the vulnerability of buffered packet hijacking on five mainstream SDN controllers including OpenDaylight, ONOS, Floodlight, RYU, and POX. Table II shows that all their newest versions have the vulnerability. Through our investigation on the reason for all mainstream SDN controllers to be vulnerable, we find the following sentence on page 111 of OpenFlow Switch Specification Version 1.5.1 (Protocol version 0x06) [22]: *"A flow mod that includes a valid buffer_id removes the corresponding packet from the buffer and processes it through the entire OpenFlow pipeline after the flow is inserted, starting at the first flow table."* Obviously, it only requires matching the buffer ID, but not the match fields.

IV. Attacks by Hijacking Buffered Packets

In this section, we design four attacks that can disrupt all layers of SDN by hijacking buffered packets. Particularly, we show how these attacks can bypass existing defense systems.

A. Cross-App Poisoning

Attack Mechanism. Figure 3 shows the cross-app poisoning attack that targets at the application layer of SDN. There are two applications concurrently running on the controller, i.e., a malicious application named as APP X and a benign application named as APP Y. When the host h_1 sends a new



```
# analyse received packet-in packets
pkt = packet.Packet(msg.data)
eth = pkt.get_protocol(ethernet)
dst = eth.dst
src = eth.src
...
#get  the datapath id
dpid = msg.datapath.id
...
#learn a mac address
self.mac_to_port[dpid][src] = msg.in_port
```

Fig. 3: Cross-APP Poisoning. A malicious application modifies and resends a buffered packet to poison other applications.

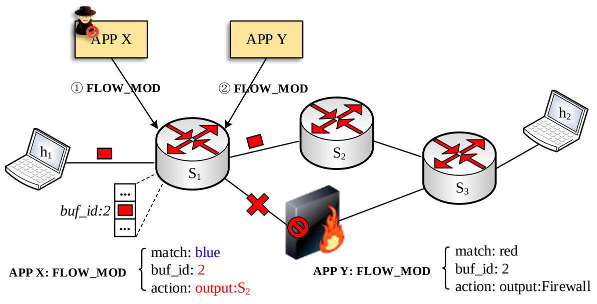


Fig. 4: Network Security Policy Bypass. A malicious application redirects buffered packets to another switch port.

flow to the host h_2, the first packet of the flow (red packet in the figure) is temporarily buffered in the switch S_1. Meanwhile, a `PACKET_IN` message is sent to the controller. APP Y is responsible for analyzing the reported packet and making decisions for the new flow. For example, it may compute routing paths and install flow rules for the new flow.

However, when APP X receives the `PACKET_IN` message before APP Y, APP X can leverage the `FLOW_MOD` message to hijack the buffered packet of the new flow. Specifically, APP X pretends to update or install flow rules for which it is responsible with the `FLOW_MOD` message. In the message, the match field is specified as the blue flow, the buffer ID is specified as the ID of the buffered red packet, and the action is specified as `set-field` and `output:controller`. In this way, APP X can manipulate the header of the buffered packet and resend it to the controller with a `PACKET_IN` message. Once the message arrives at APP Y, APP Y can be poisoned due to the extraction of the falsified packet header.

Let's see a real attack case. The codes in Figure 3 are from the learning switch application in the RYU controller. APP X changes the source MAC address of the buffered packet to the MAC address of the host h_3 with the `set-field` action. Consequently, the last line of the codes in Figure 3 falsely associates the MAC address of h_3 with the switch port connecting to h_1. According to the implementation of the learning switch, the application installs flow rules to forward flows based on the mappings of MAC addresses and switch ports. Thus, any flows with a destination address to h_3 are mistakenly directed to h_1, causing a Denial-of-Service (DoS) attack. Though this example only shows the poisoning for the learning switch application, any SDN application requiring analyzing `PACKET_IN` messages is potentially poisoned by the malicious application.

Defense Evasion. Previous work has systematically studied cross-app poisoning (CAP) and provided a defense system called ProvSDN [1]. In the previous work, a malicious application manipulates *shared data objects in the control plane* to trick another privileged application into taking actions on its behalf. The attack violates information flow control (IFC) policies due to the modification on shared data objects in the controller. Thus, by tracking the history of how shared data objects are generated and modified, ProvSDN can detect IFC policy violations and prevent CAP attacks. However, our attacking method is different. Our attack modifies *buffered packets in the data plane* and makes them resend to controllers. Since our attack does not leverage shared data objects in the control plane to poison other applications, ProvSDN fails to defeat our attack.

B. Network Security Policy Bypass

Attack Mechanism. This attack targets at the data plane layer of SDN. It redirects buffered packets to a switch port to bypass security checkpoints. As shown in Figure 4, a firewall is located between the switches S_1 and S_3. It checks each flow that crosses through the two switches according to its policies. The host h_1 is an untrusted host. When h_1 sends a flow to h_2, S_1 buffers the first packet of the flow and reports a `PACKET_IN` message to the controller. Typically, APP Y installs a flow rule with a `FLOW_MOD` message to forward the flow to the switch port that connects to the firewall. According to the firewall policies indicating that h_1 cannot communicate with h_2, the flow is blocked.

Nevertheless, a malicious application can hijack buffered packets to bypass the firewall for the first packet of the flow. When the malicious application, i.e., APP X in Figure 4, receives the `PACKET_IN` message before the APP Y, it sends a `FLOW_MOD` message to add or update a flow rule in switches. The match field of the message is the packet header of another flow that should be forwarded from S_1 to S_2, e.g., a flow from a trusted host connecting to S_1. The action of the message is to forward the matching flow to S_2. However, the buffer ID of the message is specified as the buffer ID of the buffered packet of the flow from h_1. As a result, the first packet of the flow from h_1 is released and forwarded to S_2, which bypasses the inspection of the firewall. Here, the malicious application cannot install a flow rule matching the flow from h_1 to directly forward all packets to S_2 due to the security protection of existing defense systems [15], [9], [10]. In the attack, the following packets of the flow are still forwarded to the firewall and blocked due to the following flow rules installed by APP Y. It seems that the host h_1 can only leverage the first packet to transmit information to some host and bypass the firewall for one time. However, a flow rule in SDN switches will disappear after some time according to timeout settings [22]. After the flow rule disappears, h_1 can send a flow again and leverage the first packet to continue transmitting remaining information. The previous work [23] has shown how to infer the expiration time of flow rules, which can be leveraged by h_1.

Defense Evasion. To prevent network security policy bypass and potential rule conflicts between applications, several defense systems [10], [9], [15] have been provided. However, they cannot prevent network security policy bypass that is

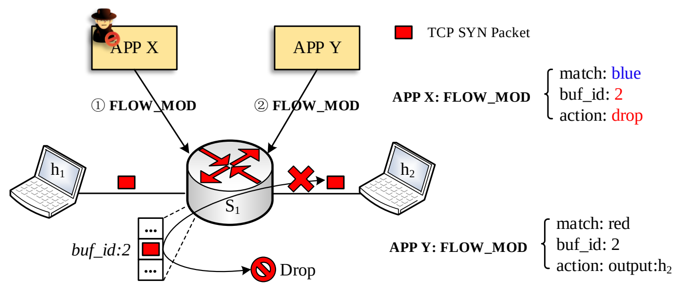


Fig. 5: TCP Three-Way Handshake Disruption. A malicious application drops a buffered TCP SYN packet.

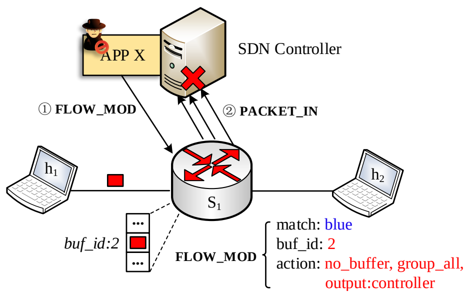


Fig. 6: Control Traffic Amplification. A malicious application copies massive buffered packets to the controller.

launched by hijacking buffered packets. Existing defense systems maintain global information of all installed flow rules and network security policies. Once an application installs or updates a flow rule in switches, the defense systems check the match fields and actions of the flow rule with advanced algorithms to see if there are conflicts with other rules and security policies. However, in our attack, the flow rule installed by the malicious application does not conflict with any rules or security policies. It is because the match field of the flow rule is not specified as the flow from the untrusted host h_1. The malicious application only uses the buffer ID of the `FLOW_MOD` message to hijack the buffered packet of the flow from h_1. We note that none of the existing defense systems check the buffer IDs. A defense named SDNShield [7] adopts a different method to prevent rule conflicts. It assigns fine-grained permission of rule installation for applications. For example, it limits an application to install flow rules for flows whose IP addresses are within 10.13.0.0/16. However, our attack still succeeds under the presence of SDNShield. Though the malicious application must install or update flow rules under the constraints of SDNShield, it can still set buffer IDs at will and thus hijacks targeted packets to bypass firewalls.

C. TCP Three-Way Handshake Disruption

Attack Mechanism. This attack targets at the data plane layer of SDN by disrupting TCP three-way handshake. As shown in Figure 5, the host h_1 aims to build a TCP connection with the host h_2 for reliable communications. According to the TCP protocol, three packets must be exchanged before a reliable TCP connection is established. The first packet is the TCP SYN packet. As there are no flow rules matching the TCP SYN packet that belongs to a new flow, the switch S_1 buffers the packet and reports a `PACKET_IN` to the controller. An application named APP Y installs flow rules with a `FLOW_MOD` message for the new flow and forwards the buffered TCP SYN packet to h_2. After that, h_2 returns a TCP packet with the SYN and ACK signal bits set to h_1. Finally, h_1 sends a TCP packet with the ACK signal bits set to finish building the TCP connection.

Similar to the attack in Figure 4, a malicious application named APP X can hijack the buffered TCP SYN packet if it receives the `PACKET_IN` message before APP Y. APP X issues a `FLOW_MOD` message to the switch S_1. The match field of the message is specified as another flow to avoid rule conflicts. The buffer ID is specified as the ID of the buffered TCP SYN packet, and the action is specified as `drop`. As a result, the buffered TCP SYN packet is dropped, which disrupts the TCP three-way handshake. After some time, h_1 will send a TCP SYN packet to h_2 again to make another try. At this time, TCP SYN packet can be successfully forwarded to h_2 since APP Y has installed related flow rules matching the packet.

Though the attack cannot completely block the TCP connections, it significantly delays the finish time of building TCP connections. According to the implementation of the network protocol stack, a second TCP SYN packet is sent after 1000 ms in the Linux operating system if a host does not get a response of the first TCP SYN packet. In windows, the time is much longer, i.e., 3000 ms. It significantly increases flow completion time (FCT) for small flows whose FCT is usually smaller than 50 ms [24]. Consequently, user experience and operator revenue are highly impacted. According to a report [25], every 100 ms latency will cost 1% in business revenue for Amazon.

Defense Evasion. As far as we know, there are no related defense considering this kind of attacks that disrupt TCP three-way handshakes by hijacking buffered packets in SDN.

D. Control Traffic Amplification

Attack Mechanism. This attack targets at the SDN control layer. It consumes bandwidth and computing resources by copying massive buffered packets to controllers. As shown in Figure 6, there are many buffered packets of new flows in switches. Switches generate `PACKET_IN` messages to the controller for rule installation. The malicious APP X receives `PACKET_IN` messages before a benign application that is responsible for installing flow rules for new flows. It allows the malicious application to hijack buffered packets. Besides hijacking buffered packets to modify packet headers or forwarding behaviors, the malicious application can copy lots of buffered packets to generate a huge amount of `PACKET_IN` messages with the `group_all` action in `FLOW_MOD` messages. The `group_all` action contains a list of action buckets. A packet is cloned for each bucket and its forwarding behavior follows the actions in a bucket. Thus, if the malicious application installs a `FLOW_MOD` message where the buffer ID is set as the ID of a buffered packet and the `group_all` action contains many buckets of the `output:controller` action, massive copies of the buffered packet will be sent to controllers with `PACKET_IN` messages. Moreover, the application can force a `PACKET_IN` message to contain the entire data packet instead of a packet header by adding the `no_buffer` action. Therefore, the attack generates an amplification effect on the control traffic since a new flow triggers more than one `PACKET_IN` message.

We define the amplification factor η as the ratio of the size of PACKET_IN messages with the attack over that without the attack. If we use n to denote the number of action buckets in a group_all action, d to denote the size of a data packet, h to denote the size of the header of the data packet, and p to denote the size of a PACKET_IN message excluding the part that stores data packets. According to the OpenFlow specification [22], a PACKET_IN message contains the first 128 bytes of a buffered data packet by default. For packets less than 128 bytes, paddings are automatically added to the message. Thus, the amplification factor η is represented as follows:

$$\eta = 1 + \frac{p + max(d, 128)}{p + 128} \cdot n \tag{1}$$

We consider a real example with a new TCP flow. The first packet of a TCP flow is always a TCP SYN packet containing no payloads, which is 74 bytes. As the packet is less than 128 bytes, the amplification factor entirely depends on the number of action buckets in a group_all action. According to our investigation, Brocade FastIron SDN switch [26] supports at most 64 action buckets in a group_all action. Thus, the amplification factor is 65. Previous studies [21], [27] have shown that the bandwidth of the control channel between a switch and a controller is tens of Mbps. Such an amplification effect on control traffic can easily make the control channel congested. Moreover, the amplification factor can further enlarge for a UDP flow since the first packet of a UDP flow is possible to reach 1518 bytes, i.e., the maximal size of a packet for Ethernet. According to our measurements, p is 108 bytes. Thus, the amplification factor can be calculated as: $1 + \frac{108+1518}{108+128} \cdot 64 \approx 442$, which is extremely large.

Defense Evasion. Previous studies [21], [27], [28], [29] have provided various defense mechanisms to mitigate potential PACKET_IN flooding attacks. They are effective to defend the flooding attack where a malicious host randomly generates anomalous packets with a high probability of matching no flow rules to trigger massive PACKET_IN messages. However, they cannot defeat the flooding attack exploiting buffered packet hijacking. Different from previous flooding attack, our attack exploits the first packet of benign data flows to trigger massive PACKET_IN messages. Thus, FloodDefender [21] fails to filter out malicious data flows and block them. FloodGuard [27] defends PACKET_IN flooding attacks by installing a wildcard flow rule of the lowest priority to forward packets matching no flow rules to data plane cache. The data plane cache schedules packets and forwards them to the controller in a rate-limited manner. However, as the flow rules installed by the malicious application has specified the actions of buffered packets, buffered packets will not match the wildcard flow rules that send packets to the data plane cache. Instead, they are copied and directly reported to controllers. Thus, our attack can flood the controller to consume bandwidth and computing resources even if FloodGuard is deployed. AVANT-GUARD [29] and LineSwitch [28] adopt the TCP SYN proxy technique to defeat the PACKET_IN flooding attack based on TCP flows. However, as mentioned by previous studies [21], [27], they are invalid for flows of other protocols. Therefore, our attack can evade them by hijacking buffered packets of other protocols, e.g., UDP packets, to flood controllers.

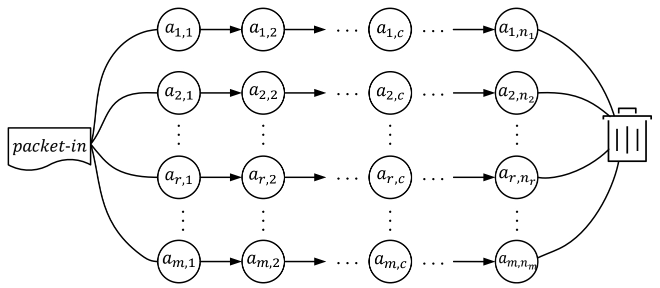


Fig. 7: The model of m processing chains. $a_{i,j}$ denotes the j-th application in the i-th processing chain with n_i applications.

V. Theoretical Analysis

In this section, we build a model of processing chains and analyze the probability of successfully launching an attack by hijacking buffered packets.

A. Processing Chain Model

The model of PACKET_IN processing chains can be illustrated in Figure 7, regardless of the implementations of different controllers. We assume there are m processing chains in a controller and n_i applications in the i-th processing chain, i.e., the length of the chain is n_i. Moreover, $a_{i,j}$ denotes the j-th SDN application in the i-th processing chain. Once a PACKET_IN message arrives at the controller, m copies of it are sent to m processing chains simultaneously. Applications get a PACKET_IN message in sequence according to their orders in a processing chain. There are many different PACKET_IN messages encapsulating different data packets. Each PACKET_IN message is processed by one or more applications that are interested in the message. An application typically continues transferring the message into the following application after finishing processing it. The last application in each processing chain stops transferring the PACKET_IN message and eliminates it.

To conduct theoretical analysis, we need to know the delay for an application to process a PACKET_IN message before it is transferred into the next application. Obviously, the processing delay is not fixed. We model the processing delay as a random variable. According to previous studies [30], the *log-logistic (fisk) distribution* has been widely used to model the processing delay when data is processed by an application and then travels to another application. Moreover, we collect a large number of real processing delays from different SDN applications. We find the log-logistic (fisk) distribution suits well for modeling the processing delays of PACKET_IN messages. For detailed distributions of processing delays, please see Appendix A.

Thus, we define the processing delay of the application $a_{i,j}$ as a random variable denoted by $D_{i,j}$. It meets the log-logistic distribution with two parameters, i.e., the scale parameter $\alpha_{i,j}$ and the shape parameter $\beta_{i,j}$. Formally, we represent it as follows:

$$D_{i,j} \sim LL(\alpha_{i,j}, \beta_{i,j}) \tag{2}$$

Here, $1 \leq i \leq m$ and $1 \leq j \leq n_i$. If we define $f_{i,j}(d)$ as the probability density function (PDF) of $D_{i,j}$, we have:

$$f_{i,j}(d) = \frac{(\beta_{i,j}/\alpha_{i,j})(x/\alpha_{i,j})^{\beta_{i,j}-1}}{(1+(x/\alpha_{i,j})^{\beta_{i,j}})^2};\ d > 0 \tag{3}$$

When $d \leq 0$, we have: $f_{i,j}(d) = 0$. We ignore the propagation delay between two successive applications for a `PACKET_IN` message since the delay is far below the processing delay in an application. For example, the propagation delay is only several microseconds according to our measurement on `Floodlight`, while the processing delay of an application is in the order of milliseconds.

B. Hijacking Probability Analysis

Based on the above model, we conduct a comprehensive analysis on the probability of successfully hijacking a buffered packet for a malicious application in processing chains. We consider that an attacker has compromised the application $a_{r,c}$ in Figure 7. The attacker aims to hijack a buffered packet that should be processed by a benign application, i.e., a target application. There are two scenarios for the malicious application to conduct the attack, i.e., attacking with an intra processing chain and attacking with inter processing chains.

Attacking with an Intra Processing Chain. To successfully launch the attack with an intra processing chain, the malicious application $a_{r,c}$ must modify buffered packets in switches before a target application modifies the buffered packets. In other words, the malicious application $a_{r,c}$ must finish processing a `PACKET_IN` message ahead of the target application. As shown in Figure 7, we can see that the attack can succeed only if the target application in the *r-th* processing chain is behind the malicious application $a_{r,c}$. If we use $a_{r,j}$ to denote the target application, the hijacking probability with the malicious application $a_{r,c}$ and the target application $a_{r,j}$ is:

$$p_{intra}(a_{r,c}, a_{r,j}) = \begin{cases} 100\%,\ if\ j \in \{1, 2, ..., c-1\} \\ 0,\ if\ j \in \{c+1, c+2, ..., n_r\} \end{cases} \tag{4}$$

As shown in Equation 4, the hijacking probability depends on the relative positions of the malicious application and the target application, regardless of processing delays.

Attacking with Inter Processing Chains. In the scenario of inter processing chains, multiple copies of a `PACKET_IN` message are fed into different processing chains. We consider that the attacker aims to hijack the buffered packet that should be processed by the target application $a_{i,j}$ in the *i-th* processing chain. If the malicious application $a_{r,c}$ can successfully modify the buffered packet in switches, the total processing delay for a `PACKET_IN` copy delivered from the first application to the malicious application in the *r-th* processing chain must be smaller than the total processing delay for a copy delivered from the first application to the *j-th* application in the *i-th* processing chain. Formally, the successful condition of hijacking the buffered packet with the malicious application $a_{r,c}$ and the target application $a_{i,j}$ can be represented as follows:

$$\sum_{k=1}^{c} D_{r,k} < \sum_{k=1}^{j} D_{i,k} \tag{5}$$

Here, $1 \leq i \leq m$ and $1 \leq j < n_i$. $D_{r,k}$ and $D_{i,k}$ are random variables meeting the log-logistic distribution. Thus, the hijacking probability with the malicious application $a_{r,c}$ and the target application $a_{i,j}$ is:

$$p_{inter}(a_{r,c}, a_{i,j}) = P(\sum_{k=1}^{c} D_{r,k} - \sum_{k=1}^{j} D_{i,k} < 0) \tag{6}$$

We now define a new random variable $Z_{i,j}$ as follows:

$$Z_{i,j} = \sum_{k=1}^{j} D_{i,k} + \sum_{k=1}^{c} (-D_{r,k}) \tag{7}$$

We define the PDF of $Z_{i,j}$ as $\hat{f}_{i,j}(z)$. We can calculate $\hat{f}_{i,j}(z)$ based on the PDF of each $D_{i,j}$ $(i \in [1,m]\ and\ j \in [1,n_i])$. Since the probability distribution of the sum of independent random variables is the convolution of their individual distributions [31], $\hat{f}_{i,j}$ can be derived as the following expression:

$$\begin{aligned} \hat{f}_{i,j}(z) = \underbrace{(\int_{-\infty}^{+\infty} \cdots \int_{-\infty}^{+\infty})}_{j+c-1} \prod_{k=1}^{j} f_{i,k}(t_k - t_{k-1}) \cdot \\ \prod_{k=1}^{c-1} f_{r,k}(t_{j+k-1} - t_{j+k}) \cdot f_{r,c}(t_{j+c-1} - z) \cdot \prod_{k=1}^{j+c-1} dt_k \end{aligned} \tag{8}$$

Here, $t_0 = 0$. Then, Equation 6 can be represented as follows:

$$\begin{aligned} p_{inter}(a_{r,c}, a_{i,j}) &= P(Z_{i,j} < 0) \\ &= \int_{-\infty}^{0} \hat{f}_{i,j}(z)dz \cdot \end{aligned} \tag{9}$$

Obviously, the above equation, i.e., p_{inter}, depends on the following variables:

$$\begin{cases} r, c, & r \in \{1, 2, ..., m\}\ and\ c \in \{1, 2, .., n_r\} \\ i, j, & i \in \{1, 2, ..., m\}\ and\ c \in \{1, 2, .., n_i\} \\ \alpha_{r,k}, \beta_{r,k}, & k \in \{1, c\} \\ \alpha_{i,k}, \beta_{i,k}, & k \in \{1, j\} \end{cases} \tag{10}$$

We can see that the hijacking probability in this scenario not only depends on the positions of the malicious and target applications, but also depends on the processing delays of other applications that are prior to the malicious application or the target application. Moreover, Equation 8 and Equation 9 show that we can calculate the numerical results of the hijacking probability for any malicious application and any target application in different positions, as long as the distribution of processing delays are measured. In the next section, we'll verify the correctness of our theoretical analysis by conducting experiments with real SDN processing chains.

VI. Attack Evaluation

In this section, we conduct experiments in a real SDN testbed to demonstrate the feasibility and effectiveness of the attacks that exploit the buffered packet hijacking vulnerability.

A. Experiment Setup

We build a real SDN testbed consisting of commercial hardware switches, EdgeCore AS4610-54T [32], and an open source SDN controller, Floodlight [17]. As we mentioned in Table II, all mainstream SDN controllers we investigated are vulnerable to buffered packet hijacking. For simplicity but

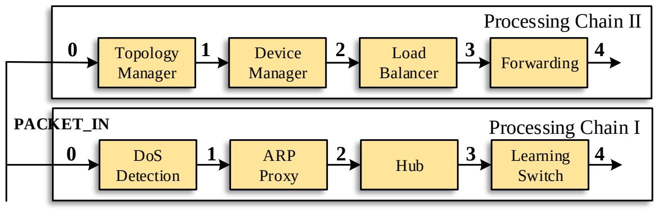


Fig. 8: The two longest processing chains in our experiments. The number is the position where a malicious application can attach to hijack buffered packets.

without loss of generality, we choose the Floodlight SDN controller to conduct our experiments since it is popular and a Java-based controller similar to most controllers. We deploy the controller on a server with a quad-core Intel Xeon CPU E5504 and 32GB RAM. We attach physical hosts to switches so as to send and receive network flows. Each host runs Ubuntu 16.04 LTS and has a quad-core Intel i3 CPU and 4GB RAM.

To build processing chains, we run all SDN applications specified in a default Floodlight configuration file, i.e., *floodlightdefault.properties* [33]. According to our analysis on the configuration file, most processing chains contain one or two applications. The longest processing chain consists of four applications, i.e., *Topology Manager* [34], *Device Manager* [35], *Load Balancer* [36], and *Forwarding* [37]. However, as we mentioned in Section V, the hijacking probability heavily depends on the positions of the malicious application and the target application. Thus, we build another long processing chain to better explore the hijacking probability for different positions of the malicious application and the target application. Specifically, we apply four applications, i.e., *DoS Detection* [38], *ARP Proxy* [39], *Hub* [40], and *Learning Switch* [41], to form the processing chain.

Figure 8 shows the two longest processing chains that contain eight SDN applications in total. The functionalities of these applications range from basic network service, such as providing network topology with *Topology Manager*, and network optimizations, such as balancing flows across multiple servers with *Load Balancer*, to advanced network security enhancement, such as detecting malicious flows with *DoS Detection*. For their detailed functionality description, we refer the readers to the links [34], [35], [36], [37], [38], [39], [40], [41]. In our experiments, we focus on the hijacking probability and attack effectiveness with the two longest processing chains. We implement and run a malicious application to hijack buffered packets with the two processing chains.

B. Hijacking Probability

To comprehensively explore the hijacking probability, we attach the malicious application in different positions of the two processing chains shown in Figure 8. Among all the eight applications in the two processing chains, only five applications, i.e., *DoS Detection*, *Hub*, *Learning Switch*, *Load Balancer*, and *Forwarding*, send `FLOW_MOD` messages to install flow rules and release buffered packets. Thus, we choose the five applications as target applications to explore the hijacking probability. To calculate the hijacking probability with a target application, we use physical hosts to generate 10,000 new flows. Each flow triggers a `PACKET_IN` message

TABLE III: Intra-Chain and Inter-Chain Hijacking Probability with Different Malicious Application's Positions

(a) When the malicious application is in Processing Chain I.

Malicious App's Position	Hijacking Probability with a Target App				
	DoS Detection	Hub	Learning Switch	Load Balancer	Forwarding
Chain I: 0	100.0%	100.0%	100.0%	100.0%	100.0%
Chain I: 1	0	100.0%	100.0%	90.0%	91.7%
Chain I: 2	0	100.0%	100.0%	70.5%	82.0%
Chain I: 3	0	0	100.0%	68.5%	80.9%
Chain I: 4	0	0	0	36.3%	57.1%
Note	**Intra-Chain Hijacking**			**Inter-Chain Hijacking**	

(b) When the malicious application is in Processing Chain II.

Malicious App's Position	Hijacking Probability with a Target App				
	Load Balancer	Forwarding	DoS Detection	Hub	Learning Switch
Chain II: 0	100.0%	100.0%	89.3%	100.0%	100.0%
Chain II: 1	100.0%	100.0%	48.8%	92.2%	95.7%
Chain II: 2	100.0%	100.0%	33.3%	85.7%	93.9%
Chain II: 3	0	100.0%	9.7%	30.6%	62.3%
Chain II: 4	0	0	8.3%	18.3%	41.9%
Note	**Intra-Chain Hijacking**		**Inter-Chain Hijacking**		

and makes the target application send `FLOW_MOD` messages to release buffered packets. Meanwhile, the malicious application also receives the PACKET_IN message and attempts to install `FLOW_MOD` messages to hijack buffered packets that should be processed by the target application. We count the number of the flows whose buffered packets are hijacked. The hijacking probability is calculated through dividing that number by the total number of flows, i.e., 10,000. Table III shows the intra-chain and inter-chain hijacking probabilities with different target applications when changing the positions of the malicious application.

Intra-chain Hijacking Probability. We first see the intra-chain hijacking probability in Table III. When the malicious application is in Processing Chain I, three target applications, i.e., *DoS Detection*, *Hub*, and *Learning Switch*, can be hijacked within the intra processing chain, which is shown in Table IIIa. When it is in Processing Chain II, another two applications, i.e., *Load Balancer* and *Forwarding*, can be hijacked within the intra processing chain, which is shown in Table IIIb. From the two tables, we can see that there are only two values of the intra-chain hijacking probability, i.e., either 100.0% or 0. As we mentioned in Section V, a malicious application can successfully hijack buffered packets with the intra processing chain if and only if it is in the front of the target application. Thus, since *DoS Detection* is the first application in Processing Chain I, the hijacking probability with it achieves 100% only when the malicious application is in the head of Processing Chain I. However, it is much easier to hijack buffered packets when the malicious application chooses *Learning Switch* or *Forwarding* as the target application. It is because both the two applications are in the tail of a processing chain. As shown in Table III, the hijacking probability with any of them is always 100.0%, except in the case where the malicious application is in the tail.

Inter-chain Hijacking Probability. It is obtained when the

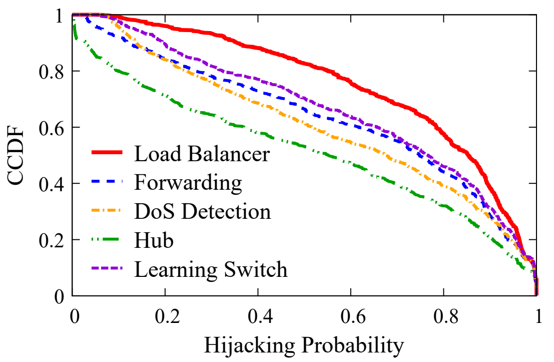


Fig. 9: CCDF of Inter-Chain Hijacking Probability with Random Positions of Target and Malicious Applications.

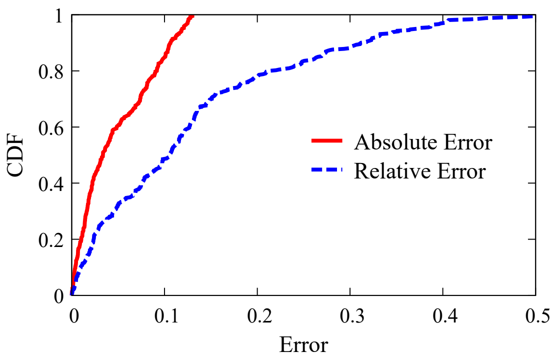


Fig. 10: Errors of Inter-Chain Hijacking Probability between Theoretical and Experimental Results.

malicious application in a processing chain attempts to hijack buffered packets that should be processed by a target application in another processing chain. Different from the intra-chain hijacking probability, it can possibly be any real numbers ranging from 0 to 100%. The inter-chain hijacking probability not only depends on relative positions of the malicious application and the target application, but also is significantly affected by processing delays of other applications in the front of the malicious or the target application. As we can see in Table IIIa and Table IIIb, the hijacking probability decreases when the malicious application moves toward the tail of a processing chain. The largest hijacking probability is 100.0% for four of the five target applications. However, the largest hijacking probability is 89.3% for *DoS Detection*. It is because *DoS Detection* is in the head of Processing Chain I. Compared to other target applications, there are more chances for the malicious application to hijack buffered packets that should be processed by *DoS Detection*. For the same reason, when the malicious application is in the tail of a processing chain, the inter-chain hijacking probability with *DoS Detection* is the smallest, i.e., 8.3%. However, the inter-chain hijacking probability with *Load Balancer*, *Forwarding*, *Hub*, and *Learning Switch* is 36.3%, 57.1%, 18.3%, and 41.9%, respectively. The above results show that the malicious application at the back has many chances to successfully hijack buffered packets that are processed by a target application at the front.

Inter-Chain Hijacking Probability with Random Positions of Target Apps. As we shown in Table III, the hijacking probability is affected by the positions of the malicious application and the target application. Particularly, the processing delays of applications in the front of the malicious or the target application have a remarkable impacts on the inter-chain hijacking probability. We implement a Java script to randomize the positions of the applications in the two processing chains [1], i.e., each application is randomly assigned to one of the eight positions in the two processing chains. Moreover, we randomly insert the malicious applications into the positions between two target applications. Figure 9 shows the Complementary Cumulative Distribution Function (CCDF) of inter-chain hijacking with random positions of target and malicious applications. Here, we do not show the CCDF of intra-chain hijacking probability since the intra-chain hijacking probability is either 0 or 100%, which is straightforward. As shown in Figure 9, the hijacking probability with any of the five applications exceeds 80% in more than 30% cases where the positions of applications are randomized. Moreover, 95% cases have a hijacking probability of more than 10% regardless of which is the target application. Above results demonstrate that a malicious application has many chances to hijack buffered packets no matter what the positions of applications are.

Errors of Inter-Chain Hijacking Probability between Theoretical and Experimental Results. By applying Equations 8 and 9 in Section V, we also calculate the theoretical inter-chain hijacking probability with different positions of target and malicious applications. We use p_t and p_e to denote the theoretical and experimental inter-chain hijacking probability, respectively. We define two types of errors, i.e., the absolute error $|p_t - p_e|$ and the relative error $\frac{|p_t - p_e|}{p_e}$. As shown in Figure 10, the absolute error is smaller than 0.1 in about 80% cases. The largest absolute error is 0.12. Moreover, the relative error is below 0.15 in about 70% cases. These results demonstrate that theoretical results are close to experimental results in most cases. However, in a few cases, the relative errors are large, although all absolute errors are small. For example, the relative errors can exceed 0.3 in about 10% cases. By analyzing the results, we find that the inter-chain hijacking probabilities in these cases are very small. Thus, even a small difference between theoretical and experimental results can lead to a large relative error. For example, we find there is an inter-chain hijacking probability of 0.011 in our experimental results, and the related theoretical probability is 0.016. We can calculate the absolute error is 0.005, which is small. However, the calculated relative error is 0.45, which is large. It is reasonable for these errors between theoretical and experimental results. As we shown in Appendix A, although the processing delays can be approximately modeled as the log-logistic distribution, there are still some differences that lead to errors.

C. Attack Effectiveness

We also conduct experiments to show the attack effectiveness of the four attacks exploiting the buffered packet hijacking vulnerability.

Cross-App Poisoning. We build the network topology in Figure 3 of Section IV. We launch an attack with the mali-

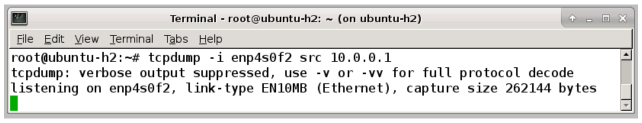
[1] We find that *Load Balancer* must be put behind *Device Manager* or be put behind *Topology Manager*. Otherwise, *Load Balancer* cannot work properly. Thus, we keep their relative orders when randomizing their positions.

cious application to poison the learning switch application by hijacking and modifying the source MAC address of a buffered packet. The learning switch application maintains mappings between MAC addresses of hosts and switch ports that connect to the hosts. It learns a mapping by analyzing the header of a data packet and the field of `in_port` in a `PACKET_IN` message. Figure 11 shows the log of the learning switch application that learns the mappings. At first, a host h_1 sends a new flow to another host h_2. The switch S_1 buffers the first packet of the flow and sends a copy of it to the controller with a `PACKET_IN` message. The learning switch application learns the host $h1$ with the MAC address *58:ef:68:13:4e:87*, which is attached to port 1 of the switch. The malicious application also receives the `PACKET_IN` message. It manipulates the source MAC address of the buffered packet to the MAC address of h_3 and makes the buffered packet resent to the controller. Thus, the learning switch application learns a false mapping between the MAC address *10:7b:44:46:e7:c1* of the host h_3 and the switch port 1, which is shown in Figure 11. Experiments show that any flow to the host h_3 is falsely directed to the host h_1 after the learning switch application is poisoned. It causes a DoS attack.

```
17:37:27.053 INFO  [n.f.l.LearningSwitch] HostMAC_to_SwitchPort =
[58:ef:68:13:4e:87=1]
17:37:49.183 INFO  [n.f.l.LearningSwitch] HostMAC_to_SwitchPort =
[58:ef:68:13:4e:87=1, 10:7b:44:46:e7:c1=1]
17:37:58.163 INFO  [n.f.l.LearningSwitch] HostMAC_to_SwitchPort =
[10:7b:44:46:e7:c1=1, 10:7b:44:46:e7:c1=1, 94:65:2d:c0:b6:96=2]
...
```

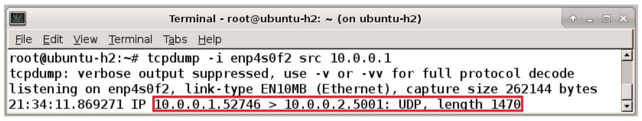
Fig. 11: The learning switch application is poisoned by the malicious application. It learns two hosts in a switch port.

Network Security Policy Bypass. We build the network topology in Figure 4 of Section IV. We use a host to implement the firewall in Figure 4. The host has two network cards that connect the switches S_1 and S_3, respectively. We enable `IP forwarding` and configure `iptables` in the host to forward packets between the two network cards. We configure `iptables` to block any packets from the host h_1 to the host h_2. We leverage the host h_1 to send a UDP packet to h_2 and launch `tcpdump` in the host h_2 to see if there is any packet from h_1. As shown in Figure 12a, the host h_2 does not receive any packet from h_1. However, when h_1 sends a UDP packet again after some time, the malicious application hijacks the buffered UDP packet and forwards it to another switch port that does not connect to the firewall. As a result, the host h_3 receives the UDP packet, which is shown in Figure 12b.


Terminal - root@ubuntu-h2: ~ (on ubuntu-h2)
File Edit View Terminal Tabs Help

```
root@ubuntu-h2:~# tcpdump -i enp4s0f2 src 10.0.0.1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on enp4s0f2, link-type EN10MB (Ethernet), capture size 262144 bytes
```

(a) When the malicious application hijacks no buffered packet


Terminal - root@ubuntu-h2: ~ (on ubuntu-h2)
File Edit View Terminal Tabs Help

```
root@ubuntu-h2:~# tcpdump -i enp4s0f2 src 10.0.0.1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on enp4s0f2, link-type EN10MB (Ethernet), capture size 262144 bytes
21:34:11.869271 IP 10.0.0.1.52746 > 10.0.0.2.5001: UDP, length 1470
```

(b) When the malicious application hijacks a buffered packet

Fig. 12: A host received a UDP packet that should have been blocked by the firewall.

TCP Three-Way Handshake Disruption. We build the network topology in Figure 5 of Section IV and launch an attack to disrupt TCP three-way handshake in the network. We use `iperf` to randomly generate many TCP flows among hosts. Our attack significantly delays establishing a TCP connection between two hosts by dropping the first TCP SYN packet. Figure 13 shows the cumulative distribution function (CDF) of connection completion time (CCT) for TCP flows. From the results, we can see that CCT for more than 90% flows is below than 10 ms without the attack. The longest CCT is about 50 ms. However, CCT of all flows is more than 1000 ms with the attack, which is twenty times longer than the longest CCT in normal cases. It is because the Linux operating system waits for one second before sending a second TCP SYN packet if the first TCP SYN packet is dropped. Such delays of TCP three-way handshake can significantly increase the flow completion time (FCT) for small flows whose FCT is typically smaller than 50 ms [24]. It inevitably degrades the user experience on using the network and the revenue for operators to provide their network services [25].

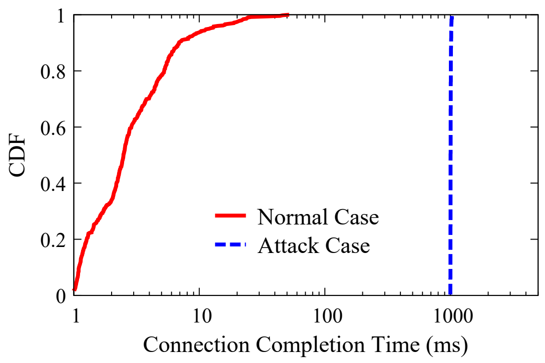
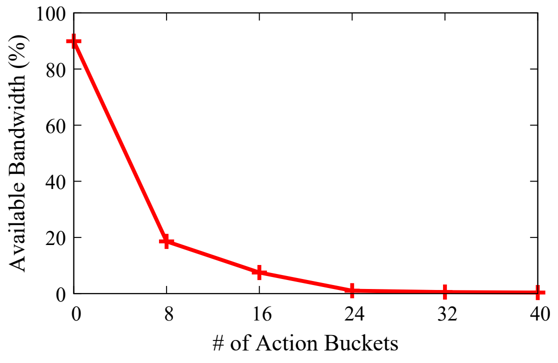


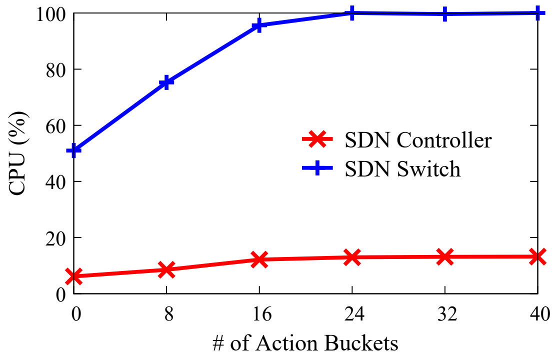
Fig. 13: Connection Completion Time (CCT) for TCP flows.

Control Traffic Amplification. We build the network topology in Figure 6. We use `TCPReplay` [42] to replay the real network traffic trace from CAIDA [43] as background flows [2]. We launch an attack that copies massive buffered packets to consume bandwidth of the SDN control channel and CPU resources of switches and controllers. Figure 14 shows the results. The number of action buckets decides the amplification factor. Figure 14a shows that the available bandwidth of the control channel is significantly decreased with the increase in the number of actions buckets. Particularly, the available bandwidth is close to 0% when there are 24 buckets, which makes new flows cannot be served. Figure 14b shows that the CPU utilization reaches 100% for an SDN switch when the number of action buckets increases to 24. However, the CPU utilization for the controller only reaches 12% when there are 24 action buckets. It is because the CPU of the host running the controller has more processing powers than switches. Moreover, since the control channel is saturated when the number of action buckets reaches 24, no more control traffic is delivered to the controller to consume its CPU resources. The CPU utilization of the controller tends to be stabilized when there are more than 24 action buckets.

[2]As the traffic trace contains a huge number of flows that can overload processing capacities of both switches and controllers, we limit the total rate of flows to less than 100 Mbps.



(a) Available Bandwidth of Control Channel



(b) CPU Utilization

Fig. 14: The effects of control traffic amplification on the bandwidth of control channel and CPU utilization.

D. Vulnerability Disclosure and Response

We reported the identified vulnerability to major SDN vendors and communities. Four SDN vendors/communities replied to us:

- *Pica8* is a mainstream SDN vendor on providing industry-leading SDN switches. They acknowledged our report and said "we have filed tracking tickets and are waiting for product management decision on releasing the fix in major/minor or patch builds".
- *ONOS* is a mainstream carrier-grade SDN controller that has been used by many service providers and enterprises. They helped us file a defect in the ONOS project and the ONOS community with the comment that "the defect will be visible to the community and this info can be available for someone to pick it up to fix it".
- *Ryu* is an open and popular SDN controller, which is maintained by the RYU community. Several developers and users in the community confirmed our report.
- *Open vSwitch* is the most popular open-source software SDN switches that enable OpenFlow. We discussed with them and they said that "OpenFlow and Open vSwitch provide no mechanisms for isolation between apps".

VII. Countermeasure

The root cause of our attacks is the missing check on the consistency between buffer IDs and match fields due to a vulnerability in the OpenFlow protocol. One intuitive countermeasure is to add the consistency checking on the SDN switches. However, it may require modifying the firmware in all SDN switches, and it is a challenge to update the firmware on various types of legacy SDN switches. Thus, we present a centralized defense solution named *ConCheck* on SDN controllers. It is lightweight and requires no modification on SDN applications and hardware. ConCheck can be easily deployed in SDN controllers to prevent a malicious application from hijacking buffered packets.

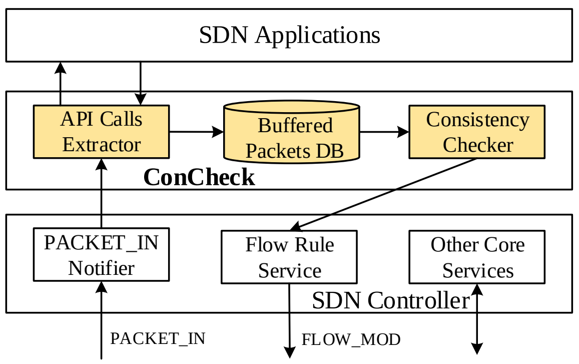


Fig. 15: The architecture of *ConCheck*.

A. Design

Figure 15 shows the architecture of ConCheck, which works between the applications and the core services of the SDN controller. The main idea is to check the consistency between the headers of buffered packets and the match fields in a `FLOW_MOD` message. When there is inconsistency, ConCheck blocks API calls that an application uses to generate flow rules. Thus, hijacking buffered packets can be prevented. ConCheck consists of two main modules: *API Calls Extractor* and *Consistency Checker*.

API Calls Extractor. To extract necessary messages for further analysis, this module intercepts API calls between SDN applications and core services. Specifically, it focuses on two types of API calls that enforce two functionalities, i.e., reading a `PACKET_IN` message and generating a `FLOW_MOD` message to install flow rules. By intercepting the first API call, the module extracts a buffer ID and the header of the buffered packet with the ID from a `PACKET_IN` message. API Calls Extractor makes a pair of them and stores the information in Buffered Packets DB for further analysis. By intercepting the second API call, it extracts a buffer ID and match fields from a `FLOW_MOD` message. The extracted information is then passed to Consistency Checker to detect potential attacks exploiting buffered packet hijacking. Meanwhile, the second API call will not be passed to flow rule service in the SDN controller until Consistency Checker checks there is no inconsistency.

Consistency Checker. This module checks if there is inconsistency for the API call that generates a `FLOW_MOD` message. As we mentioned before, applications hijacking buffered packets to disrupt SDN must install a flow rule with a `FLOW_MOD` message. However, the message contains a buffer ID of a buffered packet that matches no flow rules installed by the message. In other words, the header of the buffered packet is inconsistent with the match fields specified in the `FLOW_MOD` message. As Buffered Packets DB has stored the mapping

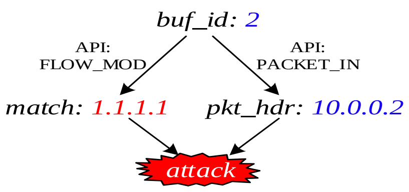


Fig. 16: An example showing ConCheck checks the inconsistency to detect attacks that hijack buffered packets.

between the buffer ID and the header of the buffered packet with that buffer ID, Consistency Checker can use the mapping to check the potential inconsistency in the API call that generates a FLOW_MOD message. Figure 16 shows an example. The API call of generating flow rules contains a buffer ID of 2 and a match field of matching packets with the IP source address of 1.1.1.1. However, from Buffered Packets DB, Consistency Checker knows that the buffered packet with the buffer ID of 2 has the IP source address of 10.0.0.2. Thus, Consistency Checker detects there is inconsistency between the header of the buffered packet and the match fields in the API call of generating flow rules. To prevent hijacking buffered packets, the API call will be blocked.

B. Evaluation

As ConCheck intercepts API calls of reading PACKET_IN and generating FLOW_MOD messages, it adds some extra delays for applications to install flow rules. Therefore, we implement a prototype of ConCheck in the Floodlight controller and measure the performance for applications to install flow rules with and without ConCheck, respectively. Figure 17 shows the CDF of the time for applications to install flow rules (flow setup time). We can see that CDF of flow setup time with ConCheck is close to that without ConCheck. More than 95% flow setup time is less than 10 ms no matter if ConCheck is deployed. Thus, ConCheck is a lightweight countermeasure and only introduces a negligible overhead when applications installing flow rules.

VIII. Related Work

There have been a number of studies [1], [11], [9], [7], [10], [18] that focus on the security threats of malicious or buggy SDN applications. Ujcich et al. [1] show cross-app poisoning attacks, in which a low privileged application tricks a high privileged application to take actions on its behalf by modifying shared data objects in controllers. They provide ProvSDN to defeat such attacks by applying data provenance and checking violations of information flow control (IFC) policies. Porras et al. [9] discover that a compromised app can manipulate flow rules to create dynamic flow tunneling, which can bypass network security polices. FortNOX [9], SE-Floodlight [10], and VeriFlow [15] detect such kinds of attacks by checking whether there are conflicts between flow rules and network security policies. Our paper provides a different attacking method for a malicious application to conduct cross-app poisoning and network security bypass attacks, i.e., exploiting the buffered packet hijacking vulnerability. Since launching attacks with the method modifies no shared data objects in controllers and generates no rule conflicts, existing countermeasures fail to defeat the attacks.

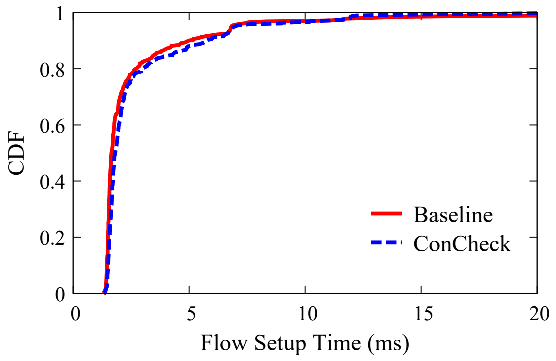


Fig. 17: CDF of flow setup time.

By abusing APIs and permissions provided by controllers, malicious applications can significantly disrupt SDN systems, such as controller rootkits [44] and SDN environment killing [11]. Therefore, researchers have provided many security enhancement systems [16], [10], [12], [7], [13], [14]. Rosemary [16] isolates applications in sandboxes to safeguard controllers from errant operations performed by applications. SE-Floodlight [10], Secure-Mode ONOS [12] and SDNShield [7] provide permission-based access control to enforce minimum required privileges to individual applications. AEGIS [13] and Controller DAC [14] enable dynamic access control for applications to protect controllers against API abuse. Previous defense systems effectively limit the SDN attack surface due to APIs and permissions abuse. However, they are not enough to prevent attacks that exploits the buffered packet hijacking vulnerability since an attacker may leverage an application that naturally has the permission of flow rule installation.

Researchers have provided a few automated analysis and test frameworks [45], [46], [47], [48] to find potential vulnerabilities in SDN applications and other components. SHIELD [45] provides an automated framework to efficiently conduct static analysis of SDN applications, which requires source codes of applications and well-defined malicious behavior to find malicious applications. Applications with unknown malicious behaviors or unavailable source codes may not be easily detected by it. DELTA [46], ATTAIN [47], and BEADS [48] aim to automatically discover new vulnerabilities resulting from applications, controllers, switches, and malicious hosts. They have effectively identified tens of new vulnerabilities. However, they do not find the buffered packet hijacking vulnerability and are necessarily incomplete due to their reliance on fuzzing.

There are some attacks generated by malicious hosts in SDN, such as topology poisoning [2], identifier binding [49], flow table saturation [23], control channel disruption [50], and PACKET_IN flooding [27]. Researchers have provided countermeasures [2], [49], [23], [50] against these attacks. Here, we focus on defense systems against the PACKET_IN flooding attack, i.e., FloodGuard [27], FloodDefender [21], AVANT-GUARD [29], and LineSwitch [28], since our paper provides a similar attack. As we detailed in Section IV, the defense systems can be evaded by our attack that triggers PACKET_IN flooding with benign flows by hijacking buffered packets rather than with a number of malicious flows.

IX. Conclusion

In this paper, we identify a new vulnerability named buffered packet hijacking that is inherent in SDN rule installation for a new flow. By exploiting the vulnerability, we discover a number of attacks that can significantly disrupt different layers of SDN and evade existing defense systems. We build a model and conduct a theoretical analysis to derive the probability of successfully hijacking a buffered packet. We evaluate the feasibility and effectiveness of the attacks in a real SDN testbed. Finally, we develop a lightweight and application-transparent countermeasure that can be readily deployed in SDN controllers as a patch.

Acknowledgment

We thank the anonymous reviewers for their constructive comments. This work is partially supported by the National Natural Science Foundation of China under Grant 61625203, 61832013, 61572278, and U1736209, the National Key R&D Program of China under Grant 2017YFB0801701, US ONR grants N00014-16-1-3214 and N00014-18-2893, US ARO grant W911NF-17-1-0447, and the US National Science Foundation (No. 1617985, No. 1642129, No. 1700544, No. 1740791). Mingwei Xu and Qi Li are corresponding authors.

References

[1] B. E. Ujcich, S. Jero, A. Edmundson, Q. Wang, R. Skowyra, J. Landry, A. Bates, W. H. Sanders, C. Nita-Rotaru, and H. Okhravi, "Cross-app poisoning in software-defined networking," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 648–663.

[2] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures." in *NDSS*, vol. 15, 2015, pp. 8–11.

[3] "Floodlight Load Balancer," https://github.com/floodlight/floodlight/tree/master/src/main/java/net/floodlightcontroller/loadbalancer, 2014, [Online].

[4] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 3–14.

[5] H. Wang, G. Yang, P. Chinprutthiwong, L. Xu, Y. Zhang, and G. Gu, "Towards fine-grained network security forensics and diagnosis in the sdn era," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 3–16.

[6] "Hewlett-Packard Enterprise: HPE SDN app store," https://community.arubanetworks.com/t5/SDN-Apps/ct-p/SDN-Apps, 2018, [Online].

[7] X. Wen, B. Yang, Y. Chen, C. Hu, Y. Wang, B. Liu, and X. Chen, "Sdnshield: Reconciliating configurable application permissions for sdn app markets," in *2016 46th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 2016, pp. 121–132.

[8] M. C. Dacier, H. König, R. Cwalinski, F. Kargl, and S. Dietrich, "Security challenges and opportunities of software-defined networking," *IEEE Security & Privacy*, vol. 15, no. 2, pp. 96–100, 2017.

[9] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for openflow networks," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 121–126.

[10] P. A. Porras, S. Cheung, M. W. Fong, K. Skinner, and V. Yegneswaran, "Securing the software defined network control layer." in *NDSS*, 2015.

[11] Lee, Seungsoo et al., "The smaller, the shrewder: A simple malicious application can kill an entire sdn environment," in *ACM SDN-NFV Security*, 2016, pp. 23–28.

[12] C. Yoon, S. Shin, P. Porras, V. Yegneswaran, H. Kang, M. Fong, B. O'Connor, and T. Vachuska, "A security-mode for carrier-grade sdn controllers," in *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACM, 2017, pp. 461–473.

[13] H. Padekar, Y. Park, H. Hu, and S.-Y. Chang, "Enabling dynamic access control for controller applications in software-defined networks," in *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies*. ACM, 2016, pp. 51–61.

[14] Y. Tseng, M. Pattaranantakul, R. He, Z. Zhang, and F. Naït-Abdesselam, "Controller dac: Securing sdn controller with dynamic access control," in *2017 IEEE International Conference on Communications (ICC)*. IEEE, 2017, pp. 1–6.

[15] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, 2013, pp. 15–27.

[16] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang, "Rosemary: A robust, secure, and high-performance network operating system," in *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. ACM, 2014, pp. 78–89.

[17] "Floodlight SDN Controller," http://www.projectfloodlight.org/floodlight/, [Online].

[18] C. Yoon and S. Lee, "Attacking sdn infrastructure: Are we ready for the next-gen networking?" in *BlackHat-USA*, 2016.

[19] J. Hizver, "Taxonomic modeling of security threats in software defined networking," in *BlackHat Conference*, 2015, pp. 1–16.

[20] "ONOS - A new carrier-grade SDN network operating system designed for high availability, performance, and scale-out," https://onosproject.org/, [Online].

[21] G. Shang, P. Zhe, X. Bin, H. Aiqun, and R. Kui, "Flooddefender: Protecting data and control plane resources under sdn-aimed dos attacks," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[22] "OpenFlow Specification v1.5.1," https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf, 2015, [Online].

[23] J. Cao, M. Xu, Q. Li, K. Sun, Y. Yang, and J. Zheng, "Disrupting sdn via the data plane: a low-rate flow table overflow attack," in *Proceedings of International Conference on Security and Privacy in Communication Systems*. Springer, 2017, pp. 356–376.

[24] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," *ACM SIGCOMM computer communication review*, vol. 41, no. 4, pp. 63–74, 2011.

[25] "Latency Is Everywhere And It Costs You Sales - How To Crush It," http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it, 2009, [Online].

[26] "FastIron Ethernet Switch Software Defined Networking: Configuration Guide," http://noc.ucsc.edu/docs/BigCreek/fastiron-08030r-manuals/fastiron-08030-sdnguide.pdf, [Online].

[27] H. Wang, L. Xu, and G. Gu, "Floodguard: A dos attack prevention extension in software-defined networks," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2015, pp. 239–250.

[28] M. Ambrosin, M. Conti, F. De Gaspari, and R. Poovendran, "Lineswitch: Tackling control plane saturation attacks in software-defined networking," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 1206–1219, 2016.

[29] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Avant-guard: Scalable and vigilant switch flow management in software-defined networks," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 413–424.

[30] "Log-logistic Distribution," https://en.wikipedia.org/wiki/Log-logistic_distribution, [Online].

[31] "Convolution of Probability Distributions," https://en.wikipedia.org/wiki/Convolution_of_probability_distributions, [Online].

[32] "AS4610-54T Data Center Switch," https://www.edge-core.com/productsInfo.php?cls=1&cls2=9&cls3=46&id=21, [Online].

[33] “Default Configuration File of Floodlight,” https://github.com/floodlight/floodlight/blob/master/src/main/resources/floodlightdefault.properties, [Online].

[34] “Floodlight Topology Manager,” https://github.com/floodlight/floodlight/tree/master/src/main/java/net/floodlightcontroller/topology/, [Online].

[35] “Floodlight Device Manager,” https://github.com/floodlight/floodlight/tree/master/src/main/java/net/floodlightcontroller/devicemanager/, [Online].

[36] “Floodlight Load Balancer,” https://github.com/floodlight/floodlight/tree/master/src/main/java/net/floodlightcontroller/loadbalancer/, [Online].

[37] “Floodlight Forwarding,” https://github.com/floodlight/floodlight/tree/master/src/main/java/net/floodlightcontroller/forwarding/, [Online].

[38] J. Zheng, Q. Li, G. Gu, J. Cao, D. K. Yau, and J. Wu, “Realtime ddos defense using cots sdn switches via adaptive correlation analysis,” *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 7, pp. 1838–1853, 2018.

[39] “Floodlight ARP Proxy,” https://github.com/mbredel/floodlight-proxyarp/, [Online].

[40] “Floodlight Hub,” https://github.com/floodlight/floodlight/tree/master/src/main/java/net/floodlightcontroller/hub/, [Online].

[41] “Floodlight Learning Switch,” https://github.com/floodlight/floodlight/tree/master/src/main/java/net/floodlightcontroller/learningswitch/, [Online].

[42] Tcpreplay - Pcap Editing and Replaying Utilities, https://tcpreplay.appneta.com/, [Online].

[43] “CAIDA Passive Monitor: Chicago B,” http://www.caida.org/data/passive/trace_stats/chicago-B/2015/?monitor=20150219-130000.UTC, [Online].

[44] C. Röpke and T. Holz, “Sdn rootkits: Subverting network operating systems of software-defined networks,” in *International Symposium on Recent Advances in Intrusion Detection*. Springer, 2015, pp. 339–356.

[45] C. Lee and S. Shin, “Shield: an automated framework for static analysis of sdn applications,” in *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*. ACM, 2016, pp. 29–34.

[46] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. A. Porras, “Delta: A security assessment framework for software-defined networks.” in *NDSS*, 2017.

[47] B. E. Ujcich, U. Thakore, and W. H. Sanders, “Attain: An attack injection framework for software-defined networking,” in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 567–578.

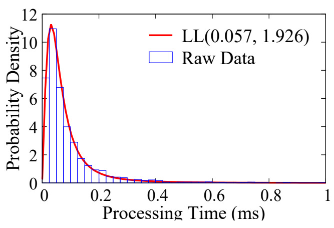
[48] S. Jero, X. Bu, C. Nita-Rotaru, H. Okhravi, R. Skowyra, and S. Fahmy, “Beads: automated attack discovery in openflow-based sdn systems,” in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017, pp. 311–333.

[49] S. Jero, W. Koch, R. Skowyra, H. Okhravi, C. Nita-Rotaru, and D. Bigelow, “Identifier binding attacks and defenses in software-defined networks,” in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 415–432.

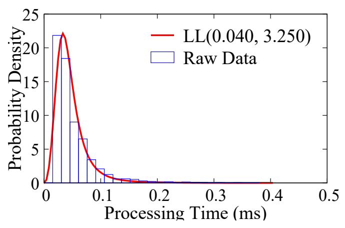
[50] C. Jiahao, L. Qi, X. Renjie, S. Kun, G. Guofei, X. Mingwei, and Y. Yuan, “The crosspath attack: Disrupting the SDN control channel via shared links,” in *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

Appendix A
Distribution of Processing Time

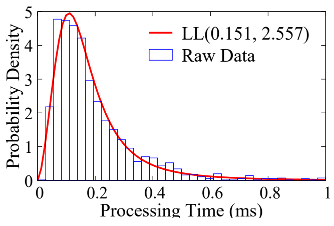
To model the processing delays of SDN applications, we collect a large number of processing delays of various SDN applications. We build a real SDN testbed consisting of commercial hardware SDN switches [32] and the `Floodlight` controller. We generate flows in the testbed to trigger `PACKET_IN` messages. We measure the processing delays of eight popular SDN applications running on the controller. They range from basic network service and network optimizations to advanced network security enhancement. For each application, we collect 1,000,000 processing delays to draw its probability density. As shown in Figure 18, the distribution of processing delays can be well modeled with the log-logistic distribution with different parameters.



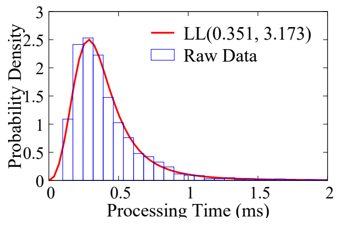
(a) Topology Manager



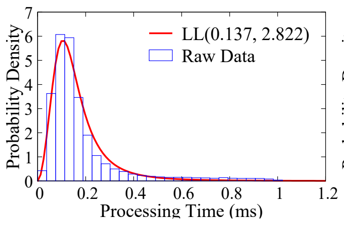
(b) Device Manager



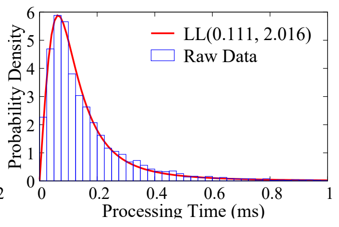
(c) Forwarding



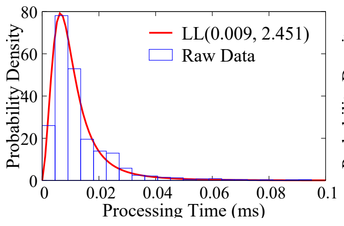
(d) Load Balancer



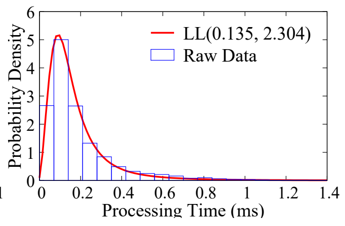
(e) Scan Detection



(f) ARP Proxy



(g) Hub



(h) Learning Switch

Fig. 18: PDF of processing delays of `PACKET_IN` messages for different SDN applications. Here, $LL(\alpha, \beta)$ denotes the log-logistic distribution with parameters α and β.