PAPER

A Packet-In Message Filtering Mechanism for Protection of Control Plane in OpenFlow Switches*

Daisuke KOTANI[†a)], ***Nonmember and*** **Yasuo OKABE**[†], ***Fellow***

SUMMARY Protecting control planes in networking hardware from high rate packets is a critical issue for networks under operation. One common approach for conventional networking hardware is to offload expensive functions onto hard-wired offload engines as ASICs. This approach is inadequate for OpenFlow networks because it restricts a certain amount of flexibility for network control that OpenFlow tries to provide. Therefore, we need a control plane protection mechanism in OpenFlow switches as a last resort, while preserving flexibility for network control. In this paper, we propose a mechanism to filter out Packet-In messages, which include packets handled by the control plane in OpenFlow networks, without dropping important ones for network control. Switches record values of packet header fields before sending Packet-In messages, and filter out packets that have the same values as the recorded ones. The controllers set the header fields in advance whose values must be recorded, and the header fields are selected based on controller design. We have implemented and evaluated the proposed mechanism on a prototype software switch, concluding that it dramatically reduces CPU loads on switches while passes important Packet-In messages for network control.

key words: *network security, software-defined networking, OpenFlow*

1. Introduction

Such networking hardware as Ethernet switches commonly forwards most packets in high performance by ASICs, and its control software runs on low performance CPUs. The software and the CPUs consist of a control plane, which controls a network like calculating and installing routing tables. Elements that are used for processing packets according to rules installed by a control plane are called a datapath like ASICs. Some functions in networking hardware, which are designed under an assumption that they are rarely used, are implemented and executed in software; however, in many cases these functions are executed frequently. Hosts sometimes send an unexpected amount of traffic, and some of such traffic may use these functions. Some networks need to use these functions to meet network requirements, and many hosts use them at the same time. In these cases, the control plane in the networking hardware becomes overloaded, operation becomes unwieldy, and networking hardware and operators often suffer from the high loads caused by such traffic. To overcome this problem, networking hardware offloads execution of these functions onto hard-wired offload engines like ASICs, Network Processors and FPGAs [2], [3].

Recent advances in Software-Defined Networking (SDN) [4] based on OpenFlow [5] allow users other than networking hardware vendors to program how networking hardware with an OpenFlow support feature (switches hereafter) forwards packets by software on external computers. The software is called a controller, which centrally manages all the switches. The control plane in OpenFlow networks includes software that handles OpenFlow messages in the switches, and the control networks between the switches and the controller, in addition to the controllers.

In OpenFlow, packet-forwarding rules are defined per flow and the rules are called flow entries. We can set flow entries for a group of flows using wildcard fields. Packets that match flow entries are processed at high speed on the datapath, which is implemented by hard-wired offload engines like ASICs. Packets that mismatch any flow entry are processed in a pre-configured way, such as sending them to controllers through the control plane as Packet-In messages, or discarding them. When a controller receives Packet-In messages, it installs new flow entries into switches, and outputs packets in the Packet-In messages to destinations. Although it is recommended that most of flow entries are installed before receiving packets, controllers need to handle some Packet-In messages to learn values of header fields from packets, such as for MAC address learning, for multicast source detection, for ARP and broadcast handling.

This SDN trend based on OpenFlow introduces a new problem: how to protect the control plane from too many packets that bring high loads in the control plane, especially in switches, while preserving flexibility for network control provided by OpenFlow. Let us assume that a host suddenly starts to send too many packets without advance notice, and switches are configured to send packets that mismatch any flow entry to controllers as Packet-In messages. There is a small time lag in switches between sending a Packet-In message and adding new flow entries that correspond to it, and a switch tries to send all mismatched packets to controllers as Packet-In messages until new flow entries are installed. As a result, a switch may be overloaded by creating and sending many Packet-In messages, delaying the handling of OpenFlow messages from the controllers. If the datapath limits the rate of packets that go to the CPU of the swtich for reducing the CPU load, packets from a host may occupy the low bandwidth between the datapath and the CPU, and the datapath may discard packets from other hosts, which

Manuscript received June 30, 2015.
Manuscript revised November 7, 2015.
Manuscript publicized December 9, 2015.

[†]The authors are with Kyoto University, Kyoto-shi, 606–8501 Japan.

[*]An earlier version of this paper was presented at ACM/IEEE ANCS 2014 [1].

a) E-mail: kotani@net.ist.i.kyoto-u.ac.jp

DOI: 10.1587/transinf.2015EDP7256

Copyright © 2016 The Institute of Electronics, Information and Communication Engineers

include important packets for controllers.

A conventional way mitigates this problem by offloading more packet processing to hard-wired offload engines. However, this choice is unsuitable to SDN based on OpenFlow because programming these engines is very different from developing software, and this difference restricts the flexibility provided by OpenFlow. Many OpenFlow extension proposals try to extend uses cases of OpenFlow [6]–[12], but it is unpractical that a switch supports all the extensions for all the potential cases. For this reason, we believe that the switches need a generic control plane protection mechanism as a last resort.

In this paper, we propose a mechanism where the switches pass important Packet-In messages for network control and apply restrictions on less important Packet-In messages. The restrictions are selected by network operators or controller developers, like filtering out the Packet-In messages or greatly limiting a Packet-In message rate.

The most important work of controllers is to continue to control networks. This work includes learning necessary state for network control, inserting and deleting flow entries quickly, etc. To continue this work, switches must not drop important packets for network control from which the controllers learn the necessary state, while the switches should drop packets that bring overloads in the control plane, especially in the switches with low CPU performance.

Although switches send entire packets as Packet-In messages, controllers use only a part of packet header values, and it depends on controller design that which header fields are used. The controllers usually parse packets and store the values of necessary header fields in the packets. If two or more packets have the same values in the necessary header fields, one packet is enough for the controllers to learn values, and the others are not necessary.

A basic approach of the proposed mechanism is simple: switches record values of header fields when they send Packet-In messages, and apply predefined restrictions on subsequent packets that match recorded entries for a certain time. To avoid an explosion of the number of entries the switches record, controllers set the switches the header fields that the controllers use, before the switches start to handle packets. The switches record only values of the header fields that the controllers have specified, and other fields are set to be wildcard.

The proposed mechanism consists of two parts: Pending Flow Rules and Pending Flow Tables. A pending flow rule is an entry to specify the header fields to which controllers refer, and each rule consists of a condition to match and a list of header fields to record. The pending flow rules are assumed to be set before switches start to handle packets. A pending flow table is a list of entries where switches record values of header fields in packets included in Packet-In messages, and we call its entries pending flow entries. One pending flow table per pending flow rule is used. A pending flow entry consists of a condition to match, and the condition is set by copying the values of the specified header fields from packets.

We implemented the proposed mechanism on Open vSwitch by extending OpenFlow's standard mechanisms. We discussed how the proposed mechanism can be used in various cases and showed that it introduces little inflexibility. We also experimentally showed that it dramatically reduced the CPU utilization in switches while forwarding important packets to controllers.

This paper is organized as follows. Section 2 introduces related work. We explain the proposed mechanism in Sect. 3, and its implementation to Open vSwitch in Sect. 4. In Sect. 5, we show how it can be used in various cases, and describe experiments to show that it reduced CPU utilization in switches. Section 6 discusses our proposed mechanism, and Sect. 7 concludes this paper.

2. Related Work

This section surveys OpenFlow and its related work for increasing scalability, and DoS attacks where situation we assume are categorized.

2.1 OpenFlow

OpenFlow [5], which was originally designed to give researchers and engineers an opportunity to easily test their new ideas in the networks that they usually use, defines an API of the datapath in switches for external control programs called controllers. Currently OpenFlow is considered an important protocol between the control plane and the datapath in Software-Defined Networking (SDN) [4].

In OpenFlow, packet-forwarding rules are stored in flow tables, where entries are called flow entries. The controllers manage the flow tables by OpenFlow protocols. One flow entry consists of three elements: a condition, actions, and statistics. The condition is used to match packets and consists of a priority value and the match fields, including an input port number and such values of packet header fields as source and destination Ethernet addresses. A wildcard is allowed in each field. The actions list those applied to matched packets, for example, outputting them to specified ports, sending them to controllers as Packet-In messages, and discarding them. The statistics include packet counters for matched packets, duration until entry's expiration, etc.

When a switch receives a packet, first it looks up its flow tables to find flow entries that match it. If some entries are found, the packet is processed based on the entries. If no entry is found, the switch handles the packet by a preconfigured table-miss entry, such as discarding it or sending it to controllers as a Packet-In message.

Flow entries are installed in two methods: proactive and reactive [13]. In the proactive method, a controller sets as many flow entries as possible to switches before hosts are connected so that the switches need not to send many Packet-In messages. Most flow entries are created using information provided by external systems, like a cloud management system. In the reactive method, a controller receives Packet-In messages from switches, and then creates

and installs flow entries based on the Packet-In messages. The controller sets no flow entry in advance. If a controller needs to learn network state that the controller cannot learn in advance, such as where hosts are connected, the controller needs Packet-In messages to learn it regardless of the methods the controllers use.

OpenFlow Switch Specification 1.3 [14] and later have a mechanism called a Meter that limits the rate of the packets in a group of flows. We can use a Meter as one of the restrictions to limit the rate of Packet-In messages.

2.2 Increasing Scalability for OpenFlow Networks

OpenFlow networks are centrally managed by controllers, who are a bottleneck in large OpenFlow networks because the controllers generally process more OpenFlow messages as a network becomes large. There are two research directions to use OpenFlow in large networks.

One approach increases message processing performance in the controllers. To achieve this, distributed controller platforms create logically centralized but physically distributed controllers. ONIX [16], HyperFlow [17], and ONOS [18] are examples of such platforms. These controllers share network state by distributed databases or storage, and each switch in the network communicates with some of physical controllers. This approach works well if each switch sends a few messages per second; it cannot alleviate the problem when the switches become overloaded, which is what we address.

The other approach enhances switch functionality so that switches can process more packets by themselves. DIFANE [12] added topology discovery and flow entry distribution functions to switches to easily enforce network access policies. DevoFlow [19] shows that the switches have meager flow setup performance, and proposes a flow entry clone flag in the flow entries mainly for elephant flow detection. The flow entry clone flag shows that, when a packet matches the flow entry, the switch creates and installs a new flow entry by copying the same actions as the matched flow entry and the values of all the header fields in the packet to the match fields. Other examples of extensions include Information-Centric Networking support [6], [10], flexible sampling actions [9], [11], and multiple output ports in one action [7]. These extensions are often designed for such specific use cases as access policy enforcement and elephant flow detection, but it is impractical to design, implement, and deploy new switch functions every time a new use case or extension emerges. Therefore, we need a generic control plane protection mechanism from many Packet-In messages as a last resort.

AVANT-GUARD [8], which is a work for control plane protection, mainly protects the control plane from TCP SYN flooding attacks by a SYN cookies approach [20]. In real networks, hosts often send such packets other than TCP as UDP and ARP, and such protocols can also be used for overloading the control plane.

Our previous work [21] filtered out Packet-In messages including packets that have the same values in all the header fields as those in previously transmitted packets. A switch creates an entry by cloning the values of all header fields in the packet to the match fields of new entries like DevoFlow [19]. The switch limits the rate of the Packet-In messages that match the entries created by this mechanism.

A serious limitation both in our previous work [21] and in DevoFlow [19] is that the number of entries generated by these mechanisms is rapidly increased, as defined by a pair of IP addresses and TCP/UDP/SCTP port numbers, and space to store these entries in switches like TCAM or RAM overflow easily, for example, by port scanning.

Another previous work of us [1] gave the proposed mechanism and basic evaluation results, but it does not evaluate the number of pending flow entries. In practice, the number of pending flow entries affects a load on switches, and we give such evaluation in this paper.

2.3 DoS Attacks

There are lots of works to prevent, detect, and filter Denial of Service attacks on the Internet [22], [23], like bandwidth consumption attacks. A problem of generating a lot of Packet-In messages in OpenFlow switches is a similar situation with bandwidth consumption attacks because the problem consumes most of bandwidth from the data plane to the CPU of a switch.

Although sophisticated algorithms against bandwidth consumption attacks like ACC [24] need to count packets that pass through links in some way before installing a rule to filter out DoS related packets, we cannot prevent the switches from overwhelming Packet-In messages by these algorithms. When a lot of Packet-In messages are being generated, both the CPU of a switch and the channel between the data plane and the CPU of a switch are overloaded, and a switch has almost no room to execute actions for counting and filtering out packets. Therefore, we need a way to set a rule to filter out floody Packet-In messages to the datapath without counting any of them.

3. Proposed Mechanism

In this section, we discuss which packets can be filtered out, then propose a mechanism to filter out packets that can be filtered out with an example.

3.1 Important Packet-In Messages for Network Control

To versatilely reduce loads on switches that are caused by too many Packet-In messages, we discuss which Packet-In messages are candidates for being filtered out by the switches. This classification must be done without asking controllers per packet.

The most important role of the controllers is to control networks, such as insertion and deletion of flow entries, and switches should not drop packets that contain important data for control, including data used for flow entries. From

this point of view, if the controllers extract and store the same data from several packets, the controllers only need one such packet, and the switches can drop the others. The switches can also drop other packets that include only unnecessary header fields. When designing a mechanism on the switches to process Packet-In messages in this way, we need to consider the following points that current OpenFlow specifications† do not handle.

The first is that the switches must forward a packet that arrives first rather than ones that arrive second or later, which have the same values in their header fields, to the controllers so that the controllers can get necessary data as soon as possible. OpenFlow expects that the controllers set flow entries quickly when a packet matches no flow entry so that subsequent packets can be processed at the datapath. If modification of flow entries is delayed, subsequent packets are also sent as Packet-In messages, and hosts cannot start communicating quickly. This is undesired behavior.

The second one is that each controller must refer to different set of header fields in packets, which depends on controller design. This means that many controllers refer to some, not all, the header fields. For example, if a controller correlates an Ethernet address with a switch and a port, it needs to learn only a source Ethernet address in from a packet. Source and destination IP addresses are adequate for a load balancing function using IP addresses.

In the following section, we explain our proposed mechanism that filters out Packet-In messages including packets that have the same values in header fields, specified by controllers, with previous Packet-In messages, while considering the header fields to which the controllers refer.

3.2 Overview of Proposed Mechanism

Figure 1 shows a conceptual diagram of an overview of our proposed mechanism. Our extensions are shown below the dotted line. We added two components to switches: Pending Flow Table and Pending Flow Rule. First, a switch looks up entries that match a packet in the flow table, then in the pending flow rule, and finally in the matched rule's pending flow table. The switch discards packets that do not match both the flow table and the pending flow rule. Packet-In messages are created when packets that do not match the flow table, match the pending flow rule, and do not match the matched rule's pending flow table.

A switch records header field values in packets into pending flow tables when the switch sends the packets to controllers as Packet-In messages. The switch regards packets that match pending flow entries as less important for network control, and apply the predefined actions to them to avoid sending them to the controllers.

The pending flow rules specify the header fields whose values are recorded in the pending flow tables. One pending flow rule includes a list of the header fields whose values are recorded and a pending flow table with entries generated

†The latest version is 1.5.1 [15] at the time of writing.

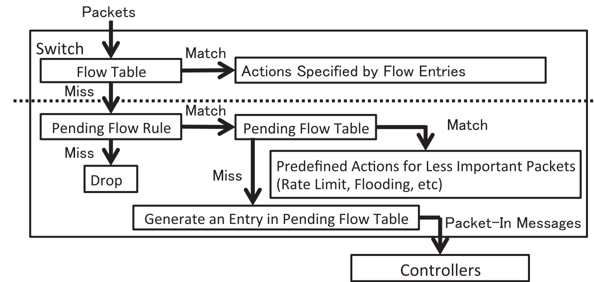

Fig. 1 Conceptual diagram for the proposed mechanism.

by the rule. The header fields in the list include those to which controllers refer, and the controllers set pending flow rules before a switch starts to process packets. If a pending flow rule matches a packet but the switch cannot find any matched pending flow entry in the associated table, the switch creates an pending flow entry from the matched rule and the packet, and installs it into the table.

The predefined actions, which are applied to less important packets, must be executed at the datapath to reduce the amount of packets handled by software in switches. Network operators or controller developers select the actions based on their policies. If they want to minimize packet loss, for example, they select a packet flooding action or an action to severely limit a Packet-In message rate. If they are not concerned with a small amount of packet loss, they select a discard action.

3.3 Pending Flow Rules

With pending flow rules, controllers inform switches which header fields are important for network control. Using this information, the switches avoid explosively increasing the pending flow entries by recording values in less important header fields. Each rule consists of the following elements.

Priority Priority value of the rule.
Timeout for Rule Timeout value of the rule.
Timeout for Table Timeout value set to pending flow entries created by the rule.
Actions List of actions to apply to packets matched with the Pending Flow Table of the rule.
Match Fields List of header fields and its values for a match condition to packets.
Clone Fields List of header fields whose values are recorded in the Pending Flow Table.
Pending Flow Table List of entries where values of the header fields are recorded by the rule.
Statistics Packet counters, duration until expire, etc.

The match fields and the priority values closely resemble flow entries in OpenFlow. The match fields include values of header fields that matched packets have, and a wildcard is allowed in each field. If a packet matches multiple rules, the highest priority rule is applied.

The clone fields are header fields whose values must be recorded in switches. The clone fields include all the header

fields that are not wildcards in the match fields of the rule, in addition to some header fields that are wildcards in the match fields. When a switch creates a new pending flow entry into the table associated with the matched rule, values of header fields listed in the clone fields are copied from the matched packet to the match fields of the new entry, and the other fields are set to be wildcards.

There are two timeout values. Timeout for Rule should be long or infinite because the main attributes of the rule, the match fields and the clone fields, are determined only by controller design, and they should be static. Timeout for Table should be small as explained in Sect. 3.4.

The statistics also resemble flow entries. The statistics provide information for controllers to determine whether to execute some entry management functions, such as evicting the rules from switches when space for storing the rules in switches become full.

3.4 Pending Flow Table

A pending flow table is a list of pending flow entries to store values in header fields of packets that a switch has already sent to controllers as Packet-In messages. Since we regard the packets that match the pending flow entries as less important ones for network control, we expect that network operators pre-configure actions for these packets, called Predefined Actions in Fig. 1, so that the switch sends fewer of them to the controllers so much. The following are examples of predefined actions: limiting the rate of Packet-In messages including these packets, flooding these packets to networks, and discarding them. We assume that these actions are executed at the datapath to prevent switches from being overloaded.

A pending flow entry resembles the flow entries, and consists of the following elements.

Match Fields List of header fields, its values, and an input port number that matched packets have. A wildcard is allowed in each field.

Timeout Timeout value until entry's removal after an entry is inserted.

Statistics Packet counters, duration until expire, etc.

The contents of the match fields are generated according to the parent pending flow rule. Values of header fields included in the clone fields of the parent rule are copied to the match fields in the entry from the matched packet. The header fields are set to wildcard if the clone fields in the parent rule do not include them.

A Timeout value should be small. Packet-In messages including important packets may be lost due to several reasons that we are not trying to solve, such as queue overflow, even when the datapath filters out less important packets by the proposed mechanism. In general, hosts resend lost packets at intervals of one or more seconds if necessary. To avoid situations where controllers do not receive important packets for a long time, we need to assure that the controllers can receive important packets without being filtered out by the pending flow tables, which are retransmitted by the hosts. In addition, once the controllers successfully install the new flow entries generated by the Packet-In messages, pending flow entries corresponding to the new flow entries are not used any more. Therefore, the pending flow entries must be deleted shortly so that retransmitted packets do not match any entry, and unused entries are not left for a long time.

The statistics are the same as the flow entries and used for the same purposes as the pending flow rules.

Entries are added when a packet matches a pending flow rule but does not match any entry in the matched rule's pending flow table, and entries are removed when timed out.

3.5 An Example of Pending Flow Rule and Table

Table 1 shows an example of pending flow rules and a pending flow entry generated by a packets and the rules. The rules are designed for a controller that manages hosts by IP addresses, and a switch forwards IPv4 packets by IPv4 addresses and other packets by Ethernet addresses.

Rule 1 is for controllers to learn hosts' IP addresses, Ethernet addresses, and port numbers. A switch records an input port number, a Frame Type (0x0800, IPv4), and source and destination Ethernet and IP addresses in the Rule 1's pending flow table. A controller sets flow entries using these parameters to forward packets by IP addresses. Rule 2 is for other packets and does not record values of IPv4 related header fields (Frame Type and IP addresses). These rules do not automatically expire, and the pending flow entries expire one second after insertion (Timeout of Rule-Clone in Table 1). The switch limits the rate of Packet-In messages including packets that match the pending flow entry.

When Packet in Table 1 arrives at a switch and no flow entry matches it, the switch looks up the pending flow rules, and both Rules 1 and 2 match the Packet. The switch chooses Rule 1 because of a higher priority value than Rule 2. Then, the switch creates a pending flow entry for Rule 1 by copying values of header fields marked "YES" in "Rule 1 - Clone" from the Packet and the Timeout for Table from

Table 1 An example of a packet, pending flow rules, pending flow tables (in part). Rules 1 and 2 match packet, and rule 1 is selected because of a higher priority value.

		Priority	In Port	Frame Type	Src MAC Addr	Dest MAC Addr	Src IP Addr	Dest IP Addr	Actions	Timeout
Packet		-	1	0x0800	00:00:5e:00:53:01	00:00:5e:00:53:02	192.0.2.1	203.0.113.1	-	-
Rule 1	Match	2	*	0x0800	*	*	*	*	-	∞
	Clone	-	Yes	Yes	Yes	Yes	Yes	Yes	Packet-In with Rate Limit	1 sec
	Table	-	1	0x0800	00:00:5e:00:53:01	00:00:5e:00:53:02	192.0.2.1	203.0.113.1	Packet-In with Rate Limit	1 sec
Rule 2	Match	1	*	*	*	*	*	*	-	∞
	Clone	-	Yes	No	Yes	Yes	No	No	Packet-In with Rate Limit	1 sec
	Table	-	-	-	-	-	-	-	-	-

Rule 1, and sends the Packet as a Packet-In message without any restriction. When the switch receives subsequent packets that have the same Ethernet and IP addresses as the Packet within one second after insertion of the Rule 1's pending flow entry, the switch limits the rate of creating and sending Packet-In messages (including such packets), and does not add any new pending flow entry.

When packets other than IPv4 ones miss any flow entry, the switch records their source and destination Ethernet addresses in the Rule 2's pending flow table. The rate of generating Packet-In messages by subsequent packets that have the same source and destination Ethernet addresses is restricted for one second by the Rule 2's pending flow table.

4. Prototype Switch Implementation

We implemented our proposed mechanism in Open vSwitch (commit 4ca808d). Figure 2 shows an overview of an Open vSwitch architecture and our modifications (marked in red). Open vSwitch consists of two parts: a datapath module and a userspace process. The datapath module handles packets in the kernel using the flow table cache. Packets that miss the flow table cache are passed to and handled by the userspace process. The userspace process also manages the flow table cache in the datapath module, and handles OpenFlow channels to controllers (OpenFlow Agent). Packets that miss the flow table cache in the datapath module and the flow table in the userspace process are translated to Packet-In messages in the OpenFlow Agent.

We translated the two components we added in Fig. 1, Pending Flow Rules and Pending Flow Tables, into one flow table to avoid increasing the matching overhead introduced by the these components. Each flow entry corresponds to an entry in the pending flow rules or tables. The flow entries for the pending flow tables have actions to limit the rate of the packets and to send to controllers, which we have selected as predefined actions for less important packets. The

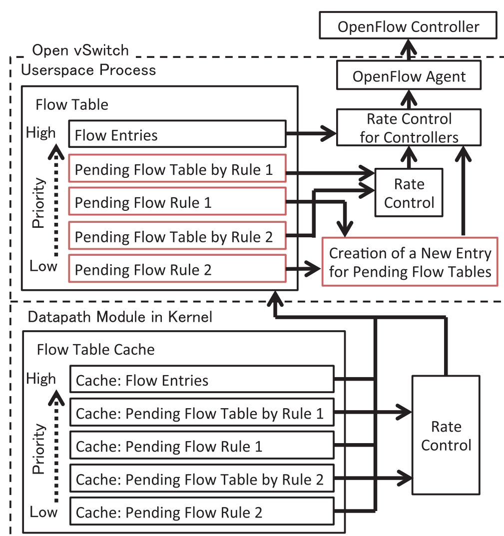

Fig. 2 A design of our prototype switch based on Open vSwitch.

flow entries for the pending flow rules have a new action we added to insert a new pending flow entry, and an action to send a Packet-In message to the controllers.

To get the same results as switches that search for the flow tables, the pending flow rules, and the pending flow tables in the order (indicated by arrows in Fig. 1), we assign different priority values to each component. We set the highest priority to the flow entries by the standard OpenFlow, and assign the lower priority to the pending flow table and rule sets. The sets of a pending flow table and rule are arranged in the order of the priorities in the pending flow rules. Within the set, a higher priority is assigned to the table, and the rule has a lower priority.

In Fig. 2, Pending Flow Rule 1 has a higher priority than Rule 2. In the flow table, the flow entries for the pending flow table of Rule 1 have a higher priority than the flow entry for Rule 1. In the same way, priorities are assigned to the flow entries related to Rule 2.

A new action's role we added is to insert a new entry in the pending flow table associated to the rule that matches a packet. The new action has the following parameters: the clone fields explained in Sect. 3.3, timeout and priority values for the new flow entries, and a Meter ID for limiting the rate of packets that match the new flow entries. When this action is executed, a switch sets a new flow entry for the pending flow table with the priority value and the timeout value in the action, the match fields whose values are copied from a packet based on the clone fields, and the predefined actions, which is to send packets to controllers through a Meter specified by the Meter ID in our case.

Controllers can set pending flow rules like flow entries. A flow entry works as a pending flow rule if it has the new action we added, a lower priority than the flow entries for the pending flow table of the rule, and a higher priority than the pending flow tables of the rules with lower priority.

The prototype switch uses a rate limiting mechanism in two ways. One limits the total rate of the Packet-In messages, which is shown as Rate Control for Controllers in Fig. 2. All Packet-In messages sent by a switch pass through this mechanism. In original Open vSwitch, this mechanism has a queue of Packet-In messages per port to avoid dropping many packets from other ports due to many packets from one port. We modified it to use only one queue because the proposed mechanism provides a very similar function in a more generic way. The other limits the rate of packets that match pending flow entries; this is shown as the Rate Control in Fig. 2. There are two Rate Controls for the pending flow tables: one in the datapath module and the other in the userspace process. Packets that match the flow entry cache of pending flow tables go through the Rate Control in the datapath module. Other packets are processed at the userspace process, and go through the Rate Control in the userspace process if necessary.

We implemented our own simple rate limiting mechanism for Rate Control because a Meter mechanism was not implemented in Open vSwitch when we implemented the prototype switch. Our simple rate limiting mechanism can

be replaced with Meter.

5. Evaluation

We have evaluated the applicability in typical uses cases, how much the loads of switches are reduced, and the analysis of execution times of each component in our prototype switch to see how our proposed mechanism changes the processing of switches.

5.1 Use Cases and Pending Flow Rules

Controllers determine how OpenFlow networks forward packets, but they often see Ethernet addresses to emulate Ethernet switches, IP addresses to emulate routers, and TCP and UDP port numbers for Network Address and Port Translation (NAPT). We can use our proposed mechanism in these scenarios as follows, and the number of required pending flow rules is summarized in Table 2.

Ethernet Switching: A controller emulating Ethernet switches must learn an association of a port where a host is connected and its Ethernet address. It sets the flow entries that have input port numbers and source and destination Ethernet addresses in the match fields to forward packets between known hosts in the datapath. In this case, we use only one pending flow rule; the controller sets a pending flow rule that records an input port number and source and destination Ethernet addresses.

IP Routing: A controller emulating IP routers must see IP addresses in packets to associate an IP address with an Ethernet address, and it must handle ARP packets in IPv4 and Neighbor Discovery in IPv6. The controller must also provide an Ethernet switching function to forward packets other than IP. This example represents cases where a switch sees header fields of protocols in multiple layers, and we use priorities. In IP Routing case, five pending flow rules are used. We set two rules with the highest priority, which match ARP and Neighbor Discovery packets. These rules copy values in ARP headers and Neighbor Discovery related headers to the match fields of new pending flow entries. The second highest priority is assigned to two rules that match other IPv4 and IPv6 packets. These rules set both source and destination IP and Ethernet addresses to the match fields in new pending flow entries. The lowest priority is assigned to one rule that matches other Ethernet frames, and the rule is the same as the case of Ethernet Switching.

NAPT: NAPT is an example of functions where a controller sees header fields in transport protocols. Controllers with the NAPT function set one pending flow rule with the highest priority per source IP address prefix to which the NAPT function is applied. The rule matches all packets whose source IP addresses are in the range to apply the NAPT function, and the clone fields of the rule include an IP protocol number, source and destination ports, and IP and Ethernet addresses. Lower priorities are assigned to the rules that handle IP and Ethernet packets, as explained for IP Routing. A similar discussion can be applied to other cases, such as a server load balancing function.

Table 2 Number of pending flow rules in typical use cases

	# of Rules
Ethernet Switching	1
IP Routing	4 + 1 (Ethernet Switching)
NAPT	1 per source IP prefix + 5 (IP Routing)

5.2 Loads in Switches

With our prototype switch, we have evaluated how much the proposed mechanism reduces a load on a switch, how many important Packet-In messages a controller can receive, and how many pending flow entries a switch has.

To measure the above points, we simultaneously sent two kinds of packets to the switch. One emulated where many packets were sent from a host without any advance notice, and called High Rate Packets. The other emulated where a small number of packets were sent from several hosts as usual called Low Rate Packets.

To measure a load caused to the switch until new flow entries are installed, a controller used in this evaluation did not install any flow entry, and the switch continued to send Packet-In messages. This is against a normal situation, which is that new flow entries are installed soon and subsequent packets are processed at the datapath, but this evaluation scenario makes us measure the load before flow entries are installed clearly. We monitored the CPU and memory utilization, the number of pending flow entries, and the traffic to the controller every second by a process running on the switch. We also counted the number of Packet-In messages by the High and Low Rate Packets separately at the controller.

We assume that a controller forwards packets by IP addresses, and we use two kinds of pending flow rules. One is the rule whose clone fields include only Ethernet and IP addresses (Rule - Host). The other is the rule whose clone fields include all header fields including Ethernet and IP addresses, and UDP port numbers (Rule - Flow). The UDP port numbers are not necessary for the controller. For comparison, we also measured without the proposed mechanism (No Rule). No Rule also represents the load before a rule to filter out High Rate Packets are not installed when the rule is not installed immediately. We limit the total rate of Packet-In messages to 100 per second and limit the rate of Packet-In messages, which include packets that match pending flow entries, to 50 per second by the predefined actions in the proposed mechanism. A timeout of pending flow entries is one second, and the pending flow rules are not expired.

High Rate Packets consist of packets with the same source and destination IP addresses and different source and destination UDP ports, and almost all of these packets are less important for network control because we assume that the controller does not see UDP port numbers. Most of these packets match the pending flow rule of Rule - Flow, and the pending flow entries of Rule - Host. To observe effects on the loads on the switch by packet rates, we change

Table 3 Number of packet-in messages and pending flow entries per second (min / average/ max)

Rate of High Rate Packets		1000	2000	3000	4000	5000
Rule Host	High Rate Packets	30 / 34.4 / 36	32 / 33.5 / 37	30 / 33.8 / 40	29 / 34.3 / 38	31 / 34.1 / 36
	Low Rate Packets	4 / 5.0 / 5	5 / 5.0 / 5	4 / 5.0 / 6	5 / 5.0 / 5	3 / 4.9 / 6
	Pending Flow Entries	6 / 6.0 / 7	6 / 6.0 / 7	6 / 6.3 / 7	6 / 6.0 / 7	6 / 6.0 / 7
Rule Flow	High Rate Packets	99 / 99.9 / 100	98 / 99.6 / 100	100 / 100 / 100	98 / 99.7 / 100	100 / 100 / 100
	Low Rate Packets	0 / 0.1 / 1	0 / 0.4 / 2	0 / 0.0 / 0	0 / 0.3 / 2	0 / 0.0 / 0
	Pending Flow Entries	1006 / 1006.1 / 1008	2007 / 2007.6 / 2010	3013 / 3013.8 / 3018	4013 / 4016.2 / 4022	5021 / 5024.3 / 5035
No Rule	High Rate Packets	100 / 100 / 100	97 / 99.2 / 100	99 / 100 / 100	100 / 100 / 100	100 / 100 / 100
	Low Rate Packets	0 / 0.0 / 0	0 / 0.8 / 3	0 / 0.0 / 1	0 / 0.0 / 0	0 / 0.0 / 0
	Pending Flow Entries	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0

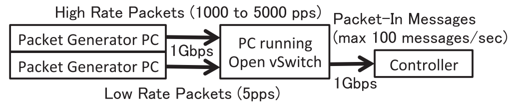

Fig. 3 Load evaluation setting

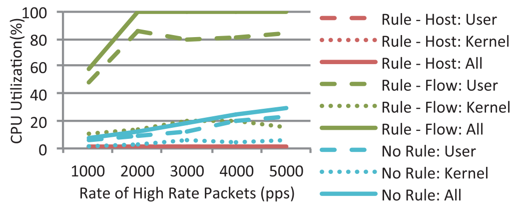

Fig. 4 CPU loads in the switch

the High Rate Packets rates to 1000, 2000, 3000, 4000, and 5000 packets per second. These rates exceed the rate limitation of Packet-In messages to the controller in the switch. Each packet in the Low Rate Packets has the same source IP address but a different destination IP address; these packets are considered important for network control. We send five Low Rate Packets per second. Both High and Low Rate Packets are sent for 30 seconds, and each packet had 128 bytes. We monitored loads on the switch and the Packet-In messages at the controller for 31 seconds including the start and end times for sending the packets, and omitted the first and last seconds from the measurement logs. The results are averages of 29 seconds.

Figure 3 shows an evaluation setting. We used four PCs, two for packet generators, one for our prototype switch, and one for an OpenFlow controller. High and Low Rate Packets were sent from different packet generator. Two packet generators and the controller were connected to the switch by 1 Gbps links separately. The packet generator had Core 2 Duo 2.16 GHz CPU, 512 MB RAM, and run Ubuntu 12.04. The switch had Xeon X5255 2.66 GHz CPU (running at 1.99 GHz, use 1 core), 8 GB RAM, and run CentOS 6.2. The controller was the same as the packet generator, but had 4 GB RAM. The maximum length of the queue from the datapath module to the userspace process was 1024 packets. We used Trema [25] and C to implement a controller monitoring Packet-In messages.

5.2.1 Results

Figure 4 shows average CPU utilization in the switch. The x-axis shows the rate of packets in High Rate Packets, and the y-axis is average CPU utilization by percentage. *User* denotes those in the userspace process, and *Kernel* means those in the datapath module. We have confirmed that CPU utilization of monitoring process was less than one percent.

When the proposed mechanism is properly configured, the CPU utilization and its increase ratio to the packet rate in both the userspace process and the datapath module are much lower in the switch with the proposed mechanism (Rule - Host) than without it (No Rule). In Rule - Flow, the CPU utilization is almost 100 percent when the rates of High Rate Packets are 2000 or more, and the figures of the Rule - Flow do not show the loads on the switches; they show the ratio of the CPU utilization of the userspace process and the datapath module, and are almost meaningless. The memory usage averaged 353 MB and did not differ among the rates of the High Rate Packets.

Table 3 shows the number of Packet-In messages per second received by the controller, and that of pending flow entries per second at the switch. *High Rate Packets* and *Low Rate Packets* show packets included in the Packet-In messages are from High or Low Rate Packets, and *Pending Flow Entries* is the number of pending flow entries in the switch.

In Rule - Flow and No Rule, which do not use the proposed mechanism properly, the packets from the High Rate Packets fill the queue of the Packet-In messages to the controller in the switch, and most of packets in the Low Rate Packets are dropped. In Rule - Host, the switch can filter out packets with a few pending flow entries. Some of the maximum number of Low Rate Packets in Rule - Host exceed five, which we sent from the packet generator as Low Rate Packets, and this should be due to a small jitter caused by the switch and the controller.

The traffic between the controller and the switch is very small compared to link speeds that the current servers and switches have. The traffic from the controller to the switch was around 10 kbps to 50 kbps, and in the opposite direction, about 70 kbps in Rule - Host, and around 180 kbps in Rule - Flow and No Rule.

5.3 Processing Time of Actions in Switches

We have also measured an execution time of each action in switches to evaluate the overhead introduced by our proposed mechanism. Major modifications in our proposed mechanism include creating an pending flow entry when a packet arrives at a switch, and limiting the rate of packets that match the entries. To evaluate how these modifications affect packet processing time, we measured the execution times related to the limitation of the rate of packets and send-

Table 4 Execution times by components in switches

Component		Execution Time in nsec (min / average / max)
Datapath	Queue to Userspace Process	3305 / 3511.9 / 6939
	Rate Limitation	75 / 201.9 / 511
	Output to Physical Port	487 / 690.8 / 4045
Userspace	Enqueue Packet-In Messages	2174 / 2440.4 / 5995
	Dequeue Packet-In Messages and Send	12621 / 13505.4 / 38641
	Dequeue Waiting Packet-In Messages and Send	10061 / 11663.9 / 20857
	Set Flow Entry for Pending Flow Table	11968 / 12617.0 / 19536

ing Packet-In messages.

In the datapath, we measured and compared the time to queue a packet to the userspace process (Queue to the Userspace Process), the rate limitation time (Rate Limit), and the time to output a packet to a port (Output Port) for reference. When the proposed mechanism is used, packets that match the pending flow entries are processed by "Rate Limit" and "Queue to the Userspace Process." Without the proposed mechanism, the packets are just processed by "Queue to the Userspace Process."

In the userspace process, we measured the time to create a flow entry for the pending flow tables (Set Filter) and to send a Packet-In message. The process of sending a Packet-In message consists of three components. First, all packets sent as Packet-In messages are queued (Enqueue Packet-In messages). Second, the packets are dequeued and sent immediately to controllers if a switch gets tokens from the rate limitation mechanism for all Packet-In messages; otherwise the packets are queued in another line (Dequeue Packet-In Messages and Send). Finally, the packets are sent when they get the tokens or are dropped if they fail to get the tokens (Dequeue waiting Packet-In Messages and Send).

We used the same PCs as Sect. 5.2. We modified the code of the prototype switch to measure the execution times per component. We executed each component 1000 times and calculated the average execution times (Table 4). The execution times for other components are not included in the results, such as flow table matching.

6. Discussion

This section discusses whether our proposed mechanism is really effective, overhead on a switch caused by our proposed mechanism, possibility to implement in hardware by comparing functions in traditional networking devices and standard OpenFlow mechanism, drawbacks of our proposed mechanism including situations where our proposed mechanism does not work well and disadvantage from the view of users, possible attacks and mitigations to our proposed mechanism, and relationship with existing attacks like DDoS and SYN floods.

6.1 Effects of Our Proposed Mechanism

As we have expected, our experiment results show that switches using the proposed mechanism properly, which use the rule that records only Ethernet and IP addresses, dramatically reduce the CPU load in the userspace process on the switches, and its benefits rise as the rate of the High Rate Packets was increased. This is because most of the High Rate Packets match pending flow entries and are filtered out at the datapath module, and the userspace process handles fewer packets than in switches without the proposed mechanism. The packet rates used in our experiment are much smaller than the maximum rate in the Gigabit Ethernet, about 1.5 million pps, but we infer from the results that these trends are the same because more packets are dropped by the datapath module if a packet rate is increased.

The results of load evaluation (Sect. 5.2) indicate that it is hard to reduce the load in the userspace process on the switches by approaches that do not install a rule to filter out High Rate Packets immediately. The load of No Rule in Fig. 4 corresponds to the load of the switches that process multiple packets in the userspace process before installing a rule to filter out such packets, like sending Packet-In messages to the controller, and counting packets for detection of DoS. The load of the userspace process in No Rule increases more than with our proposed mechanism as the rate of High Rate Packets increases, and the switches would be overloaded and become out of control before the rule for filtering out such packets is installed if a lot of packets go to the userspace process. When the switches install the rule to filter out High Rate Packets immediately like our proposed mechanism, the load of the switches would not increase so much like Rule - Host in Fig. 4.

In addition to the decrease of the load on the switches by the userspace process, the switches with our proposed mechanism can surely send Low Rate Packets to the controller and reduce the number of Packet-In messages by filtering out High Rate Packets selectively. No Rule in Table 3 indicates that High Rate Packets fill queues to send Packet-In messages to the controller, and there is no room to send Packet-In messages containing Low Rate Packets. Rule - Flow in Table 3 shows that our proposed mechanism can filter out only High Rate Packets, and the switches do not need to drop Low Rate Packets.

It depends on networks that how much Low Rate Packets are dropped in practice, such as the number of hosts and the behavior of hosts. Low Rate Packets are dropped when both High Rate Packets and Low Rate Packets are sent simultaneously. According to Table 3, almost all Low Rate Packets should be dropped at a queue from the datapath to the CPU of a switch or the userspace process in our prototype switch in this situation because the queue should be full due to High Rate Packets.

Our proposed mechanism can reduce the duration that the queue is full by High Rate Packets and Low Rate Packets are dropped. High Rate Packets are not filtered out until a flow entry to filter out such packets are installed, and Low Rate Packets may not be passed to the CPU of a switch during that time. Without our proposed mechanism, the following procedure is required to install a flow entry to filter out such packets: (1) A packet goes to the CPU of a switch from the datapath, and the software on the switch creates and sends a Packet-In message. (2) The controller sends flow entries that should be installed to the switch. (3) The

switch installs the flow entries sent by the controller in the datapath. Switches with our proposed mechanism installs an entry to filter out High Rate Packets at Step 1, and Low Rate Packets arriving after Step 1 can be passed to the controller without the effect of High Rate Packets.

According to our experiments (Table 4), the time to send a Packet-In message and to installs a flow entry to filter out High Rate Packets is in the order of microseconds. It is hard to estimate that how much it takes until a controller sends a response message to install a flow entry because it depends on controller implementation, but the time is in the order of milliseconds or tens of milliseconds according to Tootoonchian et al. [26]. Thus, in general, we can reduce the time when a switch drops Low Rate Packets by the response time of the controller.

When many pending flow entries are created like "Rule - Flow" case, the proposed mechanism works badly; almost no Low Rate Packets arrives at the controller, and a load on the switches is the highest among the three cases. This is due to a matching process of packets with a number of pending flow entries. From this results, it is recommended that controllers should not include unnecessary header fields in the clone fields of the rule to keep the number of pending flow entries low.

6.2 Overhead by Our Proposed Mechanism

Regarding the execution time overhead in the datapath, the rate limitation mechanism introduces small overhead, but the time to send packets to the userspace process is much larger than this overhead. Therefore we can reduce the CPU utilization on the datapath by limiting the rate of packets that go to the userspace process.

In the userspace process, the execution time to create and set a flow entry for the pending flow table, which includes installing a new flow entry to flow tables, is almost the same as the time to immediately send a Packet-In messages. If packets that are sent to the controller arrive at a higher rate than the limited rate, additional large execution time is needed due to queueing the packets. Although the execution time by the proposed mechanism for the first packet is twice or more than without it, it can drop many packets at the datapath and reduce the execution rate of the process to send Packet-In messages. As a result, the total execution time is much smaller with the proposed mechanism than without it when Packet-In messages are arrived at high rate. Even though the overhead evaluation is very specific to the Open vSwitch, we believe that a comparison of the execution times in the userspace process can be applied to other switches like hardware switches, because they send Packet-In messages from their software OpenFlow agents.

Another overhead introduced by the proposed mechanism is the number of pending flow entries. If they are big, large memory and TCAM space will be occupied by pending flow tables. We do not believe that this concern is significant. The pending flow rules set the header fields used for the pending flow tables based on the header fields that controllers see, and the rules should be much less than the flow entries. Both the size of the pending flow tables and the flow entries are determined by the number of different values in the header fields that the controllers see, such as IP addresses of connected hosts, and the flows sent by the hosts. The pending flow entries are soon expired, and we regard that the controllers replace the pending flow entries with the flow entries. Therefore, additional TCAM or memory space required by the pending flow tables is very small, and we can ignore this overhead unless the controllers use some flow entry compression algorithms.

Another important point is the flexibility of controlling the network. We showed that the controllers can use the proposed mechanism in typical use cases, Ethernet switching, IP routing, NAPT, etc. with a few pending flow rules. Using examples of the pending flow rules, we also showed that constructing pending flow rules is a similar process as discussing how controllers use the match fields in the flow tables. Therefore, we believe that the proposed mechanism hardly sacrifice the flexibility of controlling the network that OpenFlow provides.

6.3 Similarity with OpenFlow Mechanisms and Possibility for Hardware Implementation

The proposed mechanism should be implemented not only in software switches but also in hardware switches. The proposed mechanism's design, which has high affinity with the flow tables in OpenFlow, simplifies its implementation in hardware switches.

The matching procedures in the pending flow rules and tables are almost the same with that in the flow tables. Both use priority, input port, and the match fields for looking up their entries. Therefore, both hardware and software switches can look up entries in the pending flow rules and tables without introducing additional overhead by reusing a flow table lookup mechanism in the OpenFlow switches. In addition, the switches do not need to lookup multiple tables by translating the pending flow rules and tables to flow entries, as we did in our prototype switch.

We do not propose any specific mechanism to process packets that match the pending flow entries, and the responsibility for this mechanism, the Predefined Actions in Sect. 3.4, falls on the network operators and controller developers. We can reuse existing OpenFlow actions and instructions for the predefined actions. If they select the actions and instructions from the existing OpenFlow mechanisms, OpenFlow switches including both hardware and software switches would be able to execute the predefined actions in the datapath. For example, the prototype switch uses a combination of limiting the rate of matched packets and sending packets to the controller as Packet-In messages, which can be replaced with a Meter instruction in OpenFlow and the Output action to the controller. In Sect. 3.4, we gave another example where packets are flooded in the network. In this case, we can use a list of Output actions or the Group action to send packets to all ports in a group except the input

port. The network operators and controller developers may prefer other actions, not limited to above.

Implementation of the pending flow rules is slightly complicated. The pending flow rules require switches to create and install a new pending flow entry, including copying values of header fields specified by the clone fields from packets. It is difficult to execute this process by reusing existing OpenFlow mechanisms in the datapath, but conventional networking hardware has similar functions that create a new forwarding entry using values in packets, such as MAC address learning. We can implement actions for the pending flow rules by using these mechanisms.

The proposed mechanism is still beneficial if switches need to process an action for the pending flow rules by OpenFlow agents in the switches. After the action for the pending flow rules is executed, subsequent packets that match the new pending flow entry are filtered out in the datapath, and the load on the OpenFlow agents is reduced.

6.4 Drawbacks of Our Proposed Mechanism

The proposed mechanism works well when the datapath in switches can also extract values of packet header fields used in the match fields of the pending flow rules and tables, and when the datapath can apply the predefined actions to packets that match the pending flow entries. In other words, there might be some cases where the controllers use some header fields, but the datapath cannot parse them and extract the values. Some OpenFlow switches support part of the header fields in the OpenFlow specifications because supporting all header fields for matching is not mandatory in the specifications, for example, VLAN ID in 802.1Q headers, payloads in ARP, and ICMPv6 type and codes do not have to be parsed. We can add code to support protocols if we can write it for switches, but most hardware switches do not allow users to modify their firmware in practice.

Although OpenFlow and the proposed mechanism do not define how to handle packets that include header fields that switches cannot parse, in some cases the switches should send such packets to controllers without dropping them, for example, ARP packets. In this case, since we cannot mitigate the overloads in the switches by the proposed mechanism, we have no choice but to set the switches to send all Packet-In messages including packets of such protocols and to limit the total rate of Packet-In messages.

From the users' view, networks using the proposed mechanism may drop some packets in the beginning of such operations as connecting a new host to networks or sending a new flow. On a network side, these operations generate a new pending flow entry. For example, when a controller sets the flow entries per host and the pending flow rules whose clone fields include source and destination Ethernet and IP addresses, and a host starts to establish several TCP sessions to the same host at the same time; the second or later TCP SYN packets may be lost because of a filtering mechanism by the pending flow tables. This should not be a big problem because Ethernet and IP networks do not assure that networks deliver packets to destinations, and hosts retransmit packets at intervals of a few seconds if necessary until the hosts receive reply packets. If a certain kind of packet should not be dropped, we can use other mechanisms, like AVANT-GUARD [8] with our proposed mechanism, to provide special care for such packets.

6.5 Attacks and Mitigations

A malicious host can attack networks using OpenFlow and the proposed mechanism by exploiting it. If such attackers know header fields that controllers who manage a target network use to forward packets, they can send packets that have different values in the header fields used by the controllers. In this case, the processes for the pending flow rules are executed frequently. As a result, the CPU load on the switches is increased if the switches are implemented to execute actions for the pending flow rules in their CPUs, and the number of pending flow entries is explosively increased, like "Rule - Flow" case in the evaluation.

It is hard to distinguish these packets from others and to drop them without additional detection systems, but some mitigation exists for such attacks. Regarding the CPU load by processing the pending flow rules, controlling resources consumed by processing the rules helps keep a load low in switches at the cost of dropping packets without processing by the pending flow rules. Temporarily disabling the proposed mechanism might work well to reduce a load on switches when the switches frequently handle packets that match entries in the pending flow rules.

To keep the number of pending flow entries low, switches may have to evict some entries by cache algorithms. When no packet that increases the switch load arrives, it might not necessary to protect the switches from the overload, and the pending flow entries are merely matched with the packets; the Least Recently Used (LRU) algorithm may work well to evict the entries in this case. When switches receive attacks that explosively increase the number of pending flow entries, attackers send packets whose header fields have different values for efficient attacks, and such packets do not match any pending flow entry. The simple First In First Out (FIFO) algorithm may adequately mitigate these situations.

Finally, we discuss the relationship between the proposed mechanism and existing attacks, especially DDoS and TCP SYN floods. In these attacks, hosts send many packets to a certain network without any advance notice, and preventing switches from such packets is one of our motivations if the switches are configured to send such packets to the controllers as Packet-In messages.

It depends on controllers whether DDoS or TCP SYN flood packets can be filtered out by the proposed mechanism. When they see port numbers in packets, the clone fields in the pending flow rules include port numbers as well as IP and Ethernet addresses in the packets. In this case, the same situations happen with attacks that exploit the pending flow rules and tables, and we cannot use the proposed

mechanism to protect the switches. When controllers are designed to use wildcards in the flow entries and the packets used by the attacks only have different values in the header fields that are set to be wildcards in flow entries, we can effectively filter out such packets with the proposed mechanism and wildcards because the clone fields in the pending flow rules do not include such header fields.

7. Conclusion

In this paper, we proposed a new mechanism to protect the control plane, especially a software part in switches, from packets that bring excessive loads to it. In terms of a control plane protection, we need a mechanism to filter out less important packets for network control to keep loads on switches low in addition to extend the variety of actions that OpenFlow switches can execute.

A key of the proposed mechanism is to record values of some header fields in the switches, and the switches apply some filtering actions (predefined actions) to packets that match the recorded values. The controllers use a part of the header fields in the packets, and the switches record values of the header fields that the controllers use. The controller sets such header fields to the switches as "Pending Flow Rules", and values in packets are recorded in "Pending Flow Tables." We provide how to use the proposed mechanism in several use cases. We also provide the evaluation results using our prototype switch that show the proposed mechanism can reduce loads on switches and that important Packet-In messages are allowed to pass through. With the proposed mechanism, we can control how packets that do not match any entry in the flow tables should be handled to maintain low loads on the switches, like OpenFlow uses flow tables for offloading packet forwarding to hard-wired offload engines such as ASICs.

Future work will provide mitigation mechanisms for the explosion of the number of pending flow entries and excessive loads by attacking the pending flow rules. We only provided experimental results using a software switch, Open vSwitch. An implementation and an evaluation using hardware switches is another future work. Another future work is a way to provide flexibility of the predefined actions in the pending flow tables, such as flooding packets.

Acknowledgments

This work was supported in part by JSPS KAKENHI (No. 13J04479).

References

[1] D. Kotani and Y. Okabe, "A Packet-in Message Filtering Mechanism for Protection of Control Plane in Openflow Networks," ACM/IEEE ANCS '14, pp.29–40, ACM, Oct. 2014.

[2] W. Bux, W.E. Denzel, T. Engbersen, A. Herkersdorf, and R.P. Luijten, "Technologies and Building Blocks for Fast Packet Forwarding," IEEE Commun. Magazine, vol.39, no.1, pp.70–77, Jan. 2001.

[3] J.W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo, "NetFPGA–An Open Platform for Gigabit-Rate Network Switching and Routing," IEEE MSE '07, pp.160–161, IEEE, June 2007.

[4] N. McKeown, "Software-defined networking," IEEE INFOCOM '07, Keynote Talk, April 2009.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," SIGCOMM Comput. Commun. Rev., vol.38, no.2, pp.69–74, March 2008.

[6] D. Chang, M. Kwak, N. Choi, T. Kwon, and Y. Choi, "C-flow: An efficient content delivery framework with OpenFlow," ICOIN 2014, pp.270–275, Feb. 2014.

[7] Y. Nakagawa, K. Hyoudou, and T. Shimizu, "A Management Method of IP Multicast in Overlay Networks Using Openflow," HotSDN '12, pp.91–96, ACM, Aug. 2012.

[8] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-defined Networks," ACM CCS 2013, pp.413–424, ACM, Nov. 2013.

[9] S. Shirali-Shahreza and Y. Ganjali, "FleXam: Flexible Sampling Extension for Monitoring and Security Applications in Openflow," HotSDN '13, pp.167–168, ACM, Aug. 2013.

[10] L. Veltri, G. Morabito, S. Salsano, N. Blefari-Melazzi, and A. Detti, "Supporting Information-Centric Functionality in Software Defined Networks," IEEE ICC 2012, pp.6645–6650, June 2012.

[11] P. Wette and H. Karl, "Which Flows Are Hiding Behind My Wildcard Rule?: Adding Packet Sampling to Openflow," SIGCOMM Comput. Commun. Rev., vol.43, no.4, pp.541–542, Aug. 2013.

[12] M. Yu, J. Rexford, M.J. Freedman, and J. Wang, "Scalable Flow-Based Networking with DIFANE," ACM SIGCOMM 2010, pp.351–362, ACM, Aug. 2010.

[13] M. Fernandez, "Comparing OpenFlow Controller Paradigms Scalability: Reactive and Proactive," IEEE AINA 2013, pp.1009–1016, March 2013.

[14] Open Networking Foundation, "OpenFlow Switch Specification 1.3.4," March 2014.

[15] Open Networking Foundation, "OpenFlow Switch Specification 1.5.1," March 2015.

[16] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A Distributed Control Platform for Large-scale Production Networks," OSDI '10, pp.1–6, USENIX, Oct. 2010.

[17] A. Tootoonchian and Y. Ganjali, "HyperFlow: A Distributed Control Plane for OpenFlow," INM/WREN 2010, p.3, USENIX, April 2010.

[18] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Towards an Open, Distributed SDN OS," HotSDN '14, pp.1–6, Aug. 2014.

[19] A.R. Curtis, J.C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling Flow Management for High-Performance Networks," SIGCOMM Comput. Commun. Rev., vol.41, no.4, pp.254–265, Aug. 2011.

[20] D.J. Bernstein, "SYN cookies." http://cr.yp.to/syncookies.html

[21] D. Kotani and Y. Okabe, "Packet-In Message Control for Reducing CPU Load and Control Traffic in OpenFlow Switches," European Workshop on Software Defined Networks, pp.42–47, Oct. 2012.

[22] J. Mirkovic and P. Reiher, "A taxonomy of ddos attack and ddos defense mechanisms," SIGCOMM Comput. Commun. Rev., vol.34, no.2, pp.39–53, April 2004.

[23] T. Peng, C. Leckie, and K. Ramamohanarao, "Survey of Network-based Defense Mechanisms Countering the DoS and DDoS Problems," ACM Comput. Surv., vol.39, no.1, April 2007.

[24] R. Mahajan, S.M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker, "Controlling high bandwidth aggregates in the network," SIGCOMM Comput. Commun. Rev., vol.32, no.3, pp.62–73, July 2002.

[25] Trema Project, "Trema: Full-stack openflow framework for ruby/c."

http://trema.github.io/trema/
[26] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," Hot-ICE'12, USENIX, April 2012.

Daisuke Kotani is a student of Graduate School of Informatics, Kyoto University. He received his B.E. and M.E. degrees from Kyoto University in 2011 and 2012 respectively. His research interests include Internet architecture and distributed systems. He is a member of IPSJ, IEEE and ACM.

Yasuo Okabe received his B.E., M.E. and D.E. degrees from Department of Information Science, Kyoto University in 1986, 1988 and 1994, respectively. He was an instructor from 1988 to 1994, and was an associate professor from 1994 to 2002 at Kyoto Univesity. Since 2002 he has been a professor at Academic Center of Computing and Media Studies, Kyoto University, and he is currently the director of the center. His research interest includes Internet architecture, ubiquitous networking and distributed algorithms. He is a member of IPSJ, ISCIE, JSSST, IEEE and ACM.