

Flow consistency in an intensive SDN security architecture with multiple controllers

LV Ying-ying, GUO Yun-fei, QI Chao, WU Qi, WANG Ya-wen

(National Digital Switching System Engineering & Technological R&D Center, Zhengzhou 450002, China)

Abstract: As critical components in SDN, controllers are prone to suffer from a series of potential attacks which result in system crashes. To prevent the compromise caused by single failure of controller or flow-tampering attacks, Mcad-SA, an aware decision-making security architecture with multiple controllers was proposed, which coordinates heterogeneous controllers internally as an “associated” controller. This architecture extends existing control plane and takes advantage of various controllers’ merits to improve the difficulty and cost of probes and attacks from attackers. In this framework, flow rules distributed to switches are no longer relying on a single controller but according to the vote results from the majority of controllers, which significantly enhances the reliability of flow rules. As to the vote process of flow rules, segmentation and grading is adopted to pick up the most trustful one from multiple flow rules and implement flow consistency. This mechanism avoids comparison between rules via bit by bit which is impractical among several controllers. Theory analysis and simulation results demonstrates the effectiveness, availability and resilience of the proposed methods and their better security gain over general SDN architectures.

Key words: multi-controller, security, Mcad-SA, flow consistency

doi: 10.11959/j.issn.2096-109x.2017.00223

1 Introduction

SDN is a novel and promising architecture which separates data and control plane to facilitate the network’s management and improve its flexibility^[1]. Through introducing SDN controllers, switches no longer need handle logical judgement but convert into simple data forwarding devices while controllers take charge of sophisticated network management^[2]. However, this reliance on a centralized controller can easily result in security issues such as single point of failure. Moreover, a new set of attacks (controllers hijack, flow rules modification, etc) which are specific to SDN can compromise the network quickly if it doesn’t possess protective mechanisms. As we know, the

initial purpose of controller design is focusing on performance improvement rather than security. Thus, that how SDN should be protected is an urgent research to be conducted on^[3].

In SDN, there is no doubt that the control plane is the most important layer and responsible for generation and distribution of flow rules. Further, as key components in the control plane, controllers’ security is of great importance to guarantee the network’s operation. Moreover, due to the fact that all unknown traffic must be transmitted to controllers for investigation, it’s common that malicious traffic may lead to unpredictable attacks^[4]. Therefore, they are bound to be preferential attack targets for attackers. In other words, if we can ensure the controller’s security, then the network it

Received Data: 2017-10-11, **Revised Data:** 2017-11-08. **Corresponding Author:** LV Ying-ying, 1200062702@pku.edu.cn

Foundation Items: The National Natural Science Foundation of China (No.61521003, No.61602509), The National Key R&D Program of China (No.2016YFB0800100, No.2016YFB0800101), The Key Technologies Research and Development of Program of Henan Province (No.172102210615)

supervises will be protected wonderfully.

A typical attack in SDN is flow rule modification. For example, attackers can modify right rules into false rules they expect by means of controllers' vulnerabilities^[5]. This situation may lead to two severe consequence. One is that attackers can enable the packets forward to given hosts so they are able to capture valuable data. The other is massive packets can be forwarded to critical links, which perhaps causes these links to be congestive or paralyzed even worse. However, current defending mechanisms are ineffective to deal with this attack because there's merely one running controller. As long as it owns only a controller, the controller is certain to be compromised regardless of how powerful it is when time isn't limited resource for attackers.

In order to address above problem or alleviate its impact, we intend to solve this question from the perspective of reconstructing the control plane. Specifically, we take advantage of characteristics from various existing controllers to introduce dynamic, heterogeneity and redundancy into the control plane. That's to say, the control plane is configured with multiple controllers instead of one and its security will be significantly enhanced through coordination of all running controllers. Besides, we adopt majority decision and semantic analysis to ensure the coherence of flow rules from diverse controllers, which can guarantee rules distributed to switches are relying on vote results from operating controllers. This mechanism will further decrease the influence of single controller breakdown to some extent.

This paper is organized as follows. the next section is the related work, Section III describes the details of Mcad-SA. Section IV presents an explicit judging process of rules' coherence. In Section V, we present our evaluation results. The last section concludes by summarizing our work and discussing future work.

2 Related work

Various researches have been conducted on how to improve SDN's security and how to guarantee the coherence of flow rules when updates happen.

For the first issue, researchers attempt to address it from two aspects. On one hand, they aim at intensifying security of individual controllers via designing some internal secure mechanisms (check, verification, etc). FortNOX^[6] and SE-Floodlight^[7] are two types of typical improved controllers which are proposed recently. The former is an intensive secure NOX controller which introduces role-based authorization and security constraint enforcement to strengthen its security. It has the ability to check real-time flow-rule conflicts. Besides, it also implements a robust analysis algorithm to strategically prevent hostile applications from inserting malicious flow rules that would reduce routing efficiency in network. As to SE-Floodlight proposed by Porras, it's a security-enhanced version of the widely used Floodlight controller which extends Floodlight with a SEK (security-enforcement kernel) to primarily provide security management. The SEK consists of a particular set of secure application management policies, such as a permission model for mediating all configuration change requests to the data-plane and a security audit service. However, no matter how powerful or elaborate single controller it is, once a vulnerability is discovered or it's invaded, the security of service it provides can be no longer guaranteed. On the other hand, researchers have paid attention to distributed SDN architectures with multiple controllers which have been widely applied for failure recovery^[8] and dealing with specific attacks due to their global views, redundancy and resiliency. Typical SDN frameworks equipped with multiple controllers are ONOS^[9] and Byzantine Fault Tolerant (BFT) SDN controller^[10]. And their primary concept is introducing heterogeneity and

redundancy into the control plane to protect the network. ONOS, open network operating system, is designed to improve the robustness and resiliency of the network. It employs multiple controllers to act as different roles (master controller, backup controller, etc) in case when master controllers are unable to work, backup controllers are switched to master controllers via Zookeeper to maintain the network operate smoothly. Ref.[10] puts forward a prototype controller to tolerate byzantine faults in both the data and control plane. Even though it experiences slowdown on performance, it has the capability to tolerate a single compromised component without influencing control or forwarding decisions in the network. Although current architectures with multiple controllers can resist a series of attacks to some degree, their systems are entirely operating in a static state. However, dynamic is an effective method to improve security further. At present, dynamic is widely applied on control management^[11] than security intensification in SDN.

For the second issue, researchers mainly focus on dealing with conflicts, failures or even attacks when flow rules in switches need to be updated. In Ref.[12], Guo proposes JumpFlow to reduce flow table usage in SDN from the perspective of performance optimization when encounters updates. However, more researchers tend to put emphasis on preventing loopbacks, deadlocks and security domain breaches when updating switches. The update process is usually two-phase update. The first phase is detecting when the last packet of the old rule set has left the network and when to delete these old rules. The second phase is recoverability of the update. As to the first phase, a method is proposed in Ref.[13] to address the problem of identifying when the old rules can be deleted. While in Ref.[14], authors extend the same algorithm for per-flow consistent updates to specify when to delete the exact-match rules and consider the

failures that can occur during an update simultaneously. And enhanced algorithms are proposed to handle deletion of old rules and recoverability^[15]. Introduces an SDN verification plane to establish flow consistency between switches before flow insertion process takes place. This solution relies on a third party verification tool (UPPAAL) which translate the network view of controller to a state machine and verifies each flow before being installed. And this proposed system has the capability to enforce different levels of consistency verification whenever flow updates or topology changes occur in SDN. However, attacks aren't taken into account in above research. Ref.[16] discusses how to provide per-packet consistency in adversarial settings. Based on existing VSM (version-stamping-based mechanism)^[17], an efficient flow-ordered update mechanism is proposed to guarantee per-packet consistency against both packet-tampering and packet-dropping attacks. And it has the advantage of improving packet processing time, achieving better space efficiency and reducing the time delay for new policies to come into force. In this paper, we pay attention to another type of consistency which is concerning on how to ensure the coherence of outputs when different controllers respond to identical packet_in messages.

3 The design of Mcad-SA

In this section, the overview of Mcad-SA is illustrated in Figure 1. It's obvious that modules of the control plane have been largely extended and interactions between them are more sophisticated. Certainly, security enhancement is also benefited from the exquisite designed architecture. Overall speaking, the control plane of Mcad-SA is consisted of controller layer, management unit and proxy layer. And their respective functions are described as follows.

1) Controller layer

This layer consists of m independent running controllers, which means they don't interact with each other but receive messages from the proxy layer respectively. Meantime, they will handle the messages and generate flow rules independently. After this process, all new generated rules will be

on different hosts with various OS (Windows, Linux, MacOS, etc) to strengthen security.

2) Management unit

This module has two sub-modules. One is variant template pool and it's responsible for providing heterogenous SDN controllers for the

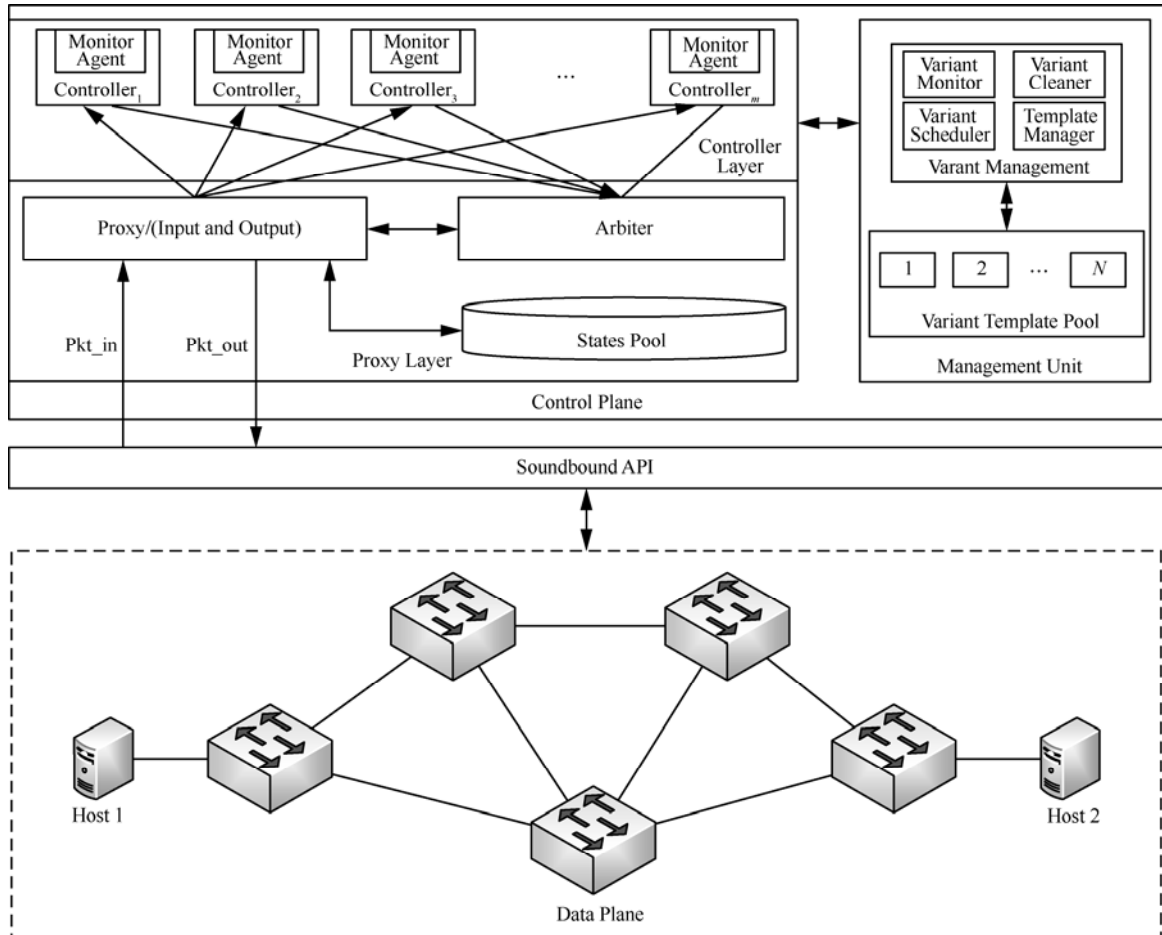


Figure 1 The overview of Mead-SA

sent back to proxy layer before distributing to switches in the data plane. Moreover, these controllers adopt heterogeneous structures (Ryu^[18], Floodlight^[19], ONOS^[9], etc.) which have individual features and are achieved with diverse programming languages. One point to be mentioned, even controllers don't belong to the same type, they will produce the identical flow rules when receiving the same information. Besides, there is a built-in monitor agent in each controller which is used for gathering running state information of controllers. Furthermore, all controllers are deployed generally

on different hosts with various OS (Windows, Linux, MacOS, etc) to strengthen security. The other is variant management and is in charge of supervising variants in that pool, including variant cleaner, variant scheduler, variant monitor and template manager. And their specific functions are presented as follows.

① Variant monitor

The primary purpose of the variant monitor is collecting real-time states of running controllers and determining whether they are benign or not. It will communicate with the monitor agent to exchange messages to analyze whether it has been probed or compromised by attackers. If so, it will produce an

00223-4

alert message and the message is sent to variant scheduler for reference.

②Variant cleaner

The main goal of the variant cleaner is cleaning, resetting and repairing running malicious variants (controllers) and ensure the variants that are put back into variant template pool are reliable.

③Variant scheduler

The vital function of variant scheduler is varying online controllers in the control layer with time according to scheduling strategies. As to the scheduling strategy, it's based on the results of variant monitor or predefined switching algorithms to pick up new reliable controllers from variant template pool to replace suspected malicious online controllers. Then these untrustworthy controllers will be handed to variant cleaner to dispose before put back to variant template pool. In general, the number of controllers selected each time is odd and changeable, which is convenient for the following decision procedure. And a typical scheduling algorithm is proposed by us which is taking maximizing the control plane's security gain as the final target^[20,21].

④Template manager

The chief duty of template manager is maintaining and supervising variants' templates in the pool, including functions of searching, deleting, insertion and instantiation. So it's able to offer robust controllers for the controller layer.

⑤Proxy layer

Its main function is acting as the interaction interface between running controllers and the data plane. Message from the data plane will be processed here before delivered to controllers. Similarly, only when flow rules produced by controllers have been handled will they be transmitted to switches. This layer has three sub-components, state pool, arbiter and input and output proxy. Next, we explain their functions in details.

⑥State pool

This pool stores up-to-date state information of running controllers. Whenever there are new online controllers, this pool has the ability to provide latest states for them to ensure they realize global information of the internet so that all running controllers can be maintained at the coherent operating states in a short time.

⑦Arbiter

This component receives flow rules from every running controller. Then it employs large number decision or relies on historical experience to determine which rules should be handed to the output proxy. Generally speaking, the primary concept of judging is based on the fact that votes from the majority of controllers are correct and reliable. In other words, it's difficult for attackers to compromise several controllers simultaneously. Therefore, the same flow rules from massive controllers can be regarded as effective, reliable and secure rules.

⑧Input and output proxy (IO Proxy)

The proxy rather than controllers communicates with underlying switches directly. That's to say, running controllers are separated from linking to switches straightly, which can protect the controller layer to some degree. When packet_in messages arrives the input proxy, the encapsulator within it first encapsulates the message, mainly inserting ID of packets, IP of controllers, and module identifiers into the header. Then the modified packet will be distributed to m online controllers in the light of IP information to process. Later, the proxy will wait for the decisions from the arbiter and dispense rules to relative switches in the underlying network after decapsulating the header. And relevant state information will be deposited in the state pool simultaneously to update data warehouse.

Finally, we concludes about the overall workflow of Mcad-SA. First, the input proxy

collects real-time packets from switches and transmits them to each running controller after encapsulating procedure. In the meantime, the built-in monitor agent keeps an eye on intrusion actions and anomaly detection of corresponding controllers. And the monitoring data is handed to the variant monitor for analyzing. If analysis results indicate there are security risk in online controllers, the variant scheduler will consider replacing skeptical controllers with authentic controllers in variant template pool. Then these offline controllers will be processed by the variant cleaner before putting back into the pool. Next, the arbiter waits for flow rules from running controllers, makes comparisons between them and chooses the most reliable rules to delivery to the output proxy. After converting rules into standardized OpenFlow protocol via decapsulating, the output proxy distributes messages to switches and updates the state pool. Ultimately, above steps will be reiterated to maintain the control plane always operate in a dynamic and unpredictable way, which enables attackers hard to probe and elevates the network's security, robustness and resilience.

4 Coherent progress of flow rules

Next, we present the details of decision procedure in the arbiter about how to guarantee the coherence of flow rules. However, in order to achieve this object, minor amendment should be adjusted to the proxy and controller layer. Moreover, the judging strategy should has the capacity to apply to various situations.

1) Problem description

As shown in Figure 2, we assume controller C_1 is under the attacker's control via attacks while the other two controllers run normally. To steal the packets from host (IP:10.1.1.1), the attacker modifies the output of the rules from port 1 to port 2. Now the packets are going to forward to switch C. Meantime, the attacker has already deployed a malicious host (IP:10.1.1.3) which is linked to switch C in advance. So it's possible for attackers to obtain what they want if the control plane is equipped with one controller, as is always the occasion in current SDN architectures.

However, this problem can be addressed perfectly in our proposed structure. In theory, the arbiter is able to take the results of "healthy controllers" as the output so rules distributed to

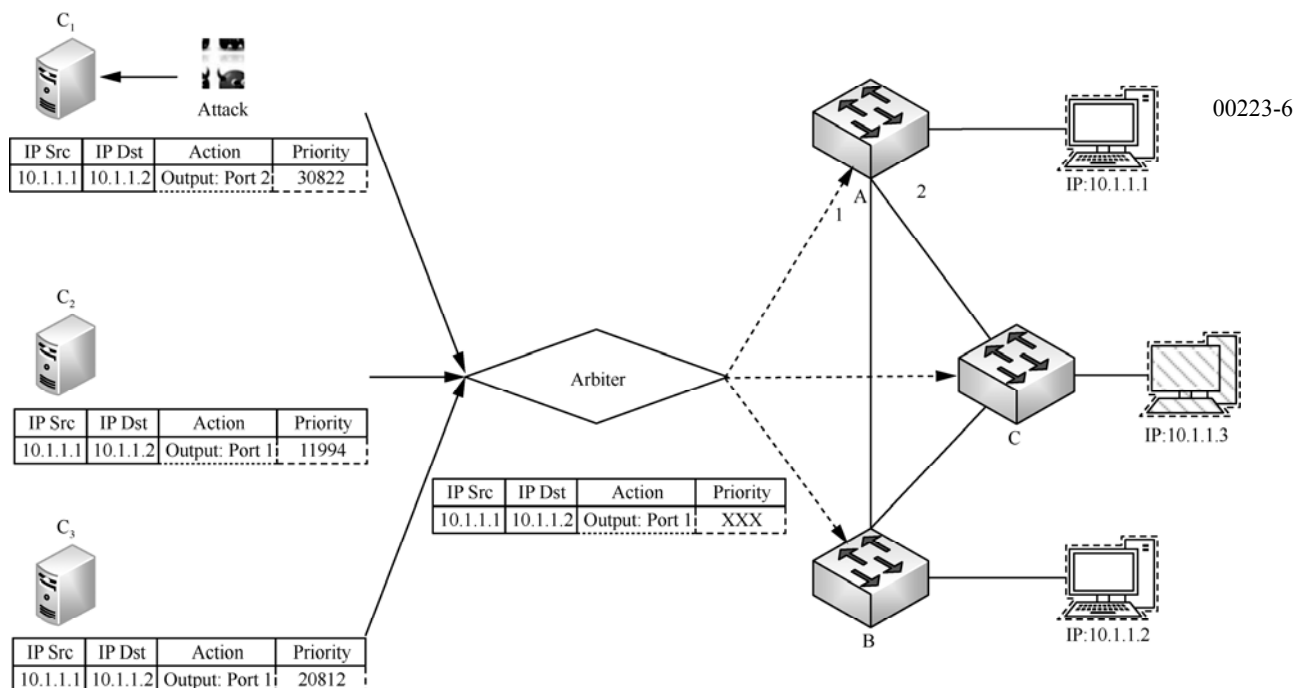


Figure 2 Flow rule attack under Mead-SA

switches are still the expectant ones and attacks on merely one controller won't have any effect on network operation. Of course, attackers may attempt to more than one controller simultaneously. But the cost is enormously increased as running controllers are alternating all the time, which is definitely painstaking for them to locate and probe them. Even though they discover the vulnerabilities of different controllers, they have to produce consistent false outputs by means of these vulnerabilities to deceive the arbiter. This target is also difficult to be implemented.

Actually, the deciding procedure in the arbiter isn't as simple as imagined. As we all know, although diverse controllers (Ryu, floodlight, etc) adopt OpenFlow protocol, their outputs are not absolutely identical when they receive the equal packet_in messages. For example, the priority of the same flow rule from floodlight and Ryu may be different, so do other fields. Thus, it's an impractical method to compare the flow rules bit by bit straightly.

In order to eliminate above impact, the arbiter adopts grading based on segment match instead of direct comparison to obtain coherent rules for switches, which can improve the efficiency of comparison and avoid false judgment. The primary conception of grading based on segment match is

breaking up rules into small pieces (such as action, priority), then large number decision is conducted on divided segments. In each segment match, if a segment from rule A is the same to the majority, then it's labeled with a high score. While that is different, it's labeled with a lower score or even zero. When all fragments have finished this process, we can obtain several scores for each rule. Then we can acquire the total score of each rule by summing them. Undoubtedly, the higher score a rule gets, the more reliable it is. And the rule with the highest score is the one we expect to hand to the proxy. One point to be pointed out, to reach above goals, the IO proxy and controller layer need minor adjustment to facilitate the judging step. Therefore, before introducing details of segment match, we present their inner compositions at first.

2) Inner compositions in IO proxy and controller layer

As is described in section II, IO proxy is consisted of encapsulator and decapsulator. Here, we further illustrates their specific implementing modes as indicated in Figure 3.

When the packet_in messages arrives at the encapsulator, it will adopt tunnel protocol to insert a header into the packet. And the header contains protocol version, type, length, packet_id and

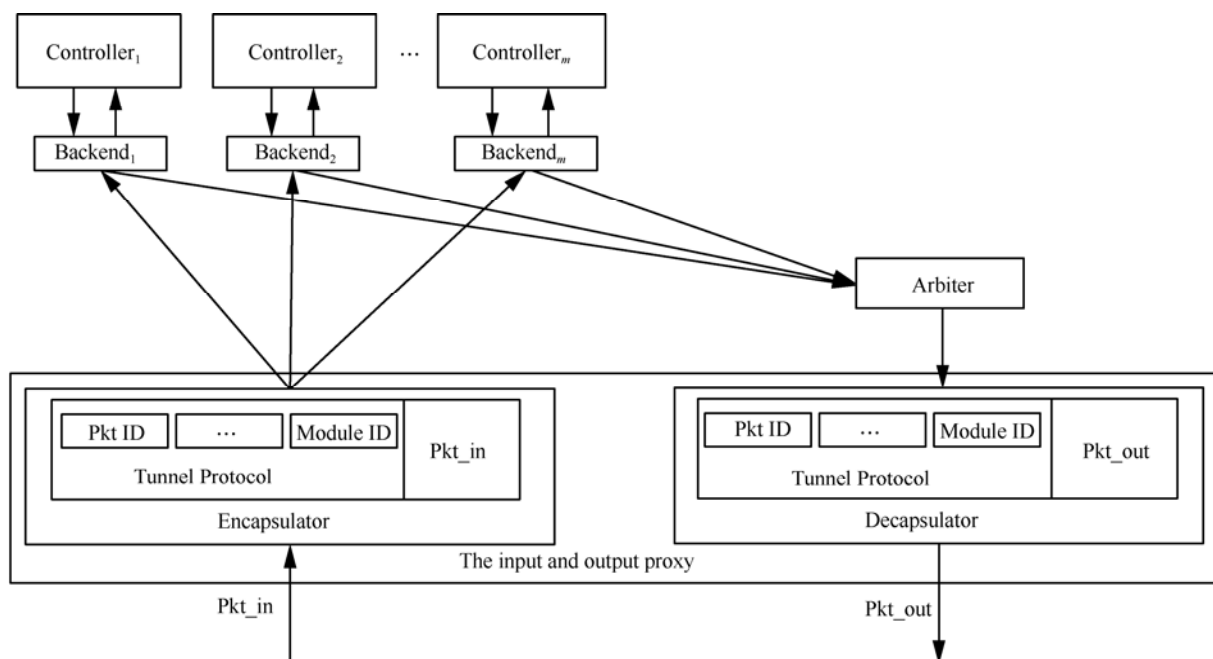


Figure 3 Inner compositions in IO proxy and controller layer

module_id. Figure 4 provides the length of each field. And each unit represent 8 bit.

0	1	2	3
version	type	length	
packet_id			
module_id			

Figure 4 The header of tunnel protocol

As to the explicit explanation of these fields, we illustrate them in Table 1.

Table 1 Notations of fields in the header	
Field	Notations
version	The version of control protocol we take
type	The type of messages
length	The length of payload
packet_id	The unique id of message
module_id	The unique id of module operating on the controller

Besides, there is a backend in front of each controller, or it can be taken as a new module we add in the controller. And its basic function is just extracting payload of the message and handing it to the rest modules of controller to handle. An advantage of this design is avoiding modifying existing modules, which brings simplicity and practicability. After the payload has been processed and new messages have been produced, the backend adds the header into the generated message again before sending to the arbiter. The reason why we reinsert the header is facilitating the following judging procedure in the arbiter. On receiving messages from the backend, the arbiter initiates the judging process which is going to be discussed in the next part. The decision result will be given to the decapsulator whose function is similar to the backend. Then it gets rid of the header and distributed the normal OpenFlow messages to the underlying switches.

3) Decision process of flow rule

As we mentioned before, that whether the arbiter can finally obtain the correct and effective

flow rules is of extreme importance to the implementation of Mcad-SA. In this part, we will present the detailed process that how we acquire the most reliable and valid flow rule among multiple messages from the backend step by step. And this mission can be decomposed into four procedures: Synchronization, Segmentation, Comparison and Decision.

①Synchronization

The purpose of synchronization is capturing all flow rules from each backend (or each controller) which are responding to the identical packet_in messages. Clearly, through matching two fields (packet_id and module_id) in the header, this object can be realized easily. And that’s the reason why we maintain the header all the time.

②Segmentation

Due to difference of flow rules produced from various controllers, it’s impossible to conduct matching bit by bit. Thus, the OpenFlow message will be split into small segmentation according to field semantics.

③Comparison

The comparison is focused on each specific segment. For different segments, the granularity of comparison is discriminatory. For example, strict match should be carried out on fields like IP source, IP destination and output port while vague match can be implemented on fields like priority and idle_time. The basis of field division is whether the value of a field in a rule is the same no matter which controller produces it when receiving the identical packet_in messages. If they are uniform, then the fields belong to the type of strict match and vice versa.In strict match, a larger value is assigned to the majority while a lower one is given to the minority. While in vague match, we neglect these difference and allocate the same value.

④Decision

When comparison in each segment is finished and the corresponding scores are obtained. We sort

the rules based on the total score in a descending order and pop up the first one as the output of the arbiter.

Take the case in Figure 2 as the example, we depict the whole decision process in Figure 5 (the synchronization step is neglected).

In above case, controller C_1 is controlled by the attacker. So it has the privilege to modify the fields of rule (R_1). To obtain desired information, the attacker alters the output port into 2 and promotes the priority to the highest. In order to remove the modified rule, each rule is decomposed into three segments (IP src and dst, action and priority) and they are put into corresponding units to be handled independently. For segment 1, three controllers' output are equal so they gain the same score (here the value is 1). While the situation is a bit different in segment 2, the action from R_1 is diverse from the rest. Therefore the score of R_1 is 0 and the rest is 1.

In segment 3, the value of priority is different from each one, but this field belongs to vague match and it's not critical to the result. Thus we set the score as 1 for all. Now we've already gotten the scores of all the segments, next these scores are summed up together. And the result is that the total score of R_1 is 2 while R_2 and R_3 are 3. This indicates that R_2 and R_3 are more reliable than R_1 . So in the final decision process, R_2 is popped up. Of course, there is a probability that the arbiter chooses R_3 eventually.

4) Time diagram of messages

Figure 6 illustrates the time diagram of messages between various components. First, the workflow starts with request messages (packet_in messages) which are put forward by network elements. When they arrive at the proxy, it computes a new unique ID of packet (packet_id = N) which is not being used in other existing transactions and inserts it into the header. Then the

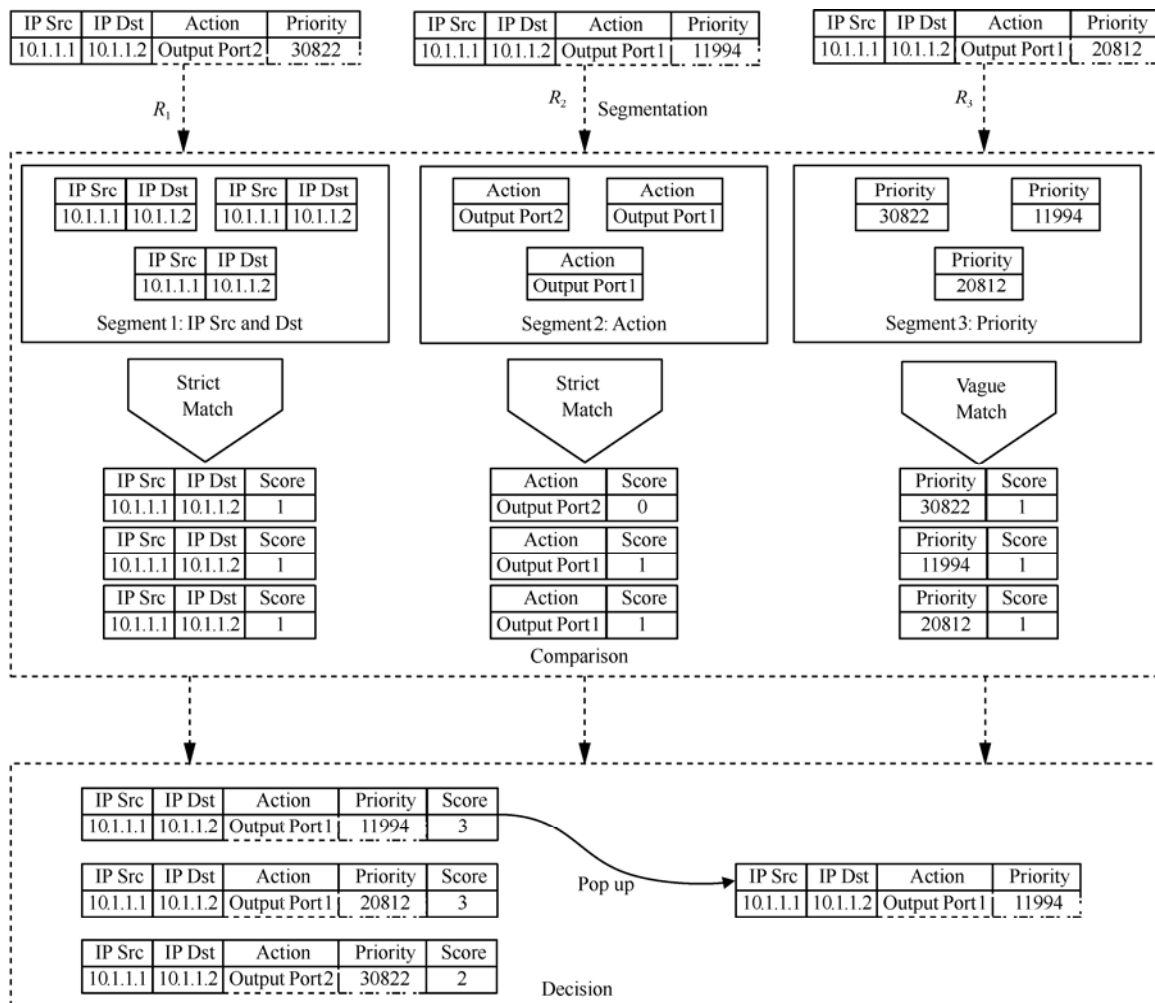


Figure 5 An instance of decision process

packets go straight to backend to remove the header before sending to specific modules. After the corresponding module has handled the message, the reply message is obtained with an ID of module ($\text{module_id} = M$). When the reply message returns to the backend, it reinserts the header which contains the packet_id . Then, reply messages continue to reach the arbiter and wait for processing. As to its decision strategy, we will discuss it in detail in the following part. Next, the proxy extracts the standardized OpenFlow messages and replies to network elements asking for requests.

As to “controller-to-switch” messages defined in the OpenFlow specifications, the OpenFlow messages generated by controllers are ignored by the arbiter. They are simply relayed to backends then forwarded to the proxy. While the other procedures are similar to that of asynchronous OpenFlow messages. At present,

backends and proxy can be regarded as forwarding equipment and don't have any logical functions.

5) Decision strategy of the arbiter

In Section C, we introduce the deciding process of multiple flow rules. However, there is one more problem to be considered. It's common that requirements of the underlying network are always varying according to specific situations. Sometimes, packet_in messages have to be responded swiftly to satisfy fast network changes. While sometimes the network doesn't have high demands in terms of timeliness. Thus, it's necessary for the arbiter to have the capability of adjusting to meet above needs. Here, we propose two decision strategies to achieve fore mentioned goals: one is AISO (all in single out), the other is FIFO (first in first out).

① AISO

This strategy is suitable for the network which

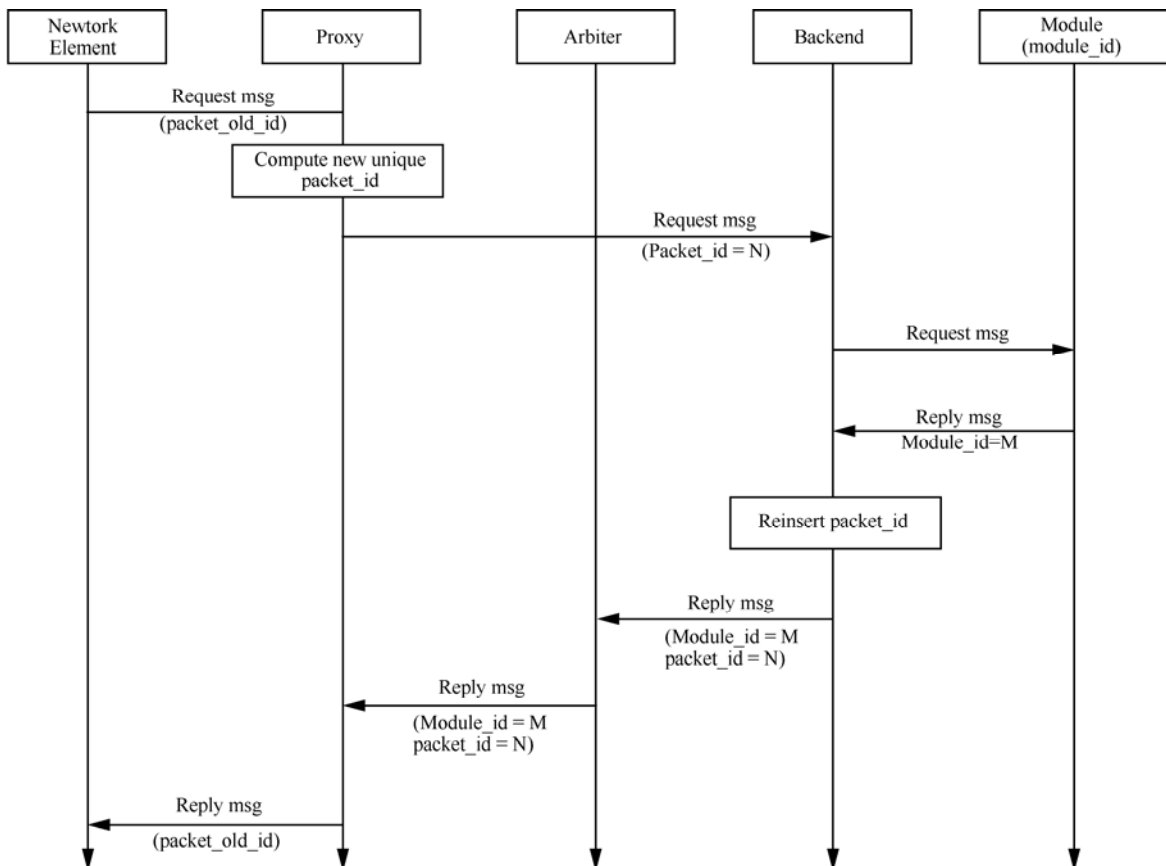


Figure 6 Time diagram of messages

can tolerate heavy latency. And the concrete steps are exhibited in Figure 7.

As messages from backends are unlikely to arrive at the arbiter simultaneously, so the message that comes first have to wait for other $m-1$ messages (given m running controllers) before the decision process. For the fake of convenience, the first arrived message (Msg_i) is put into the bottom of stack and so on. When the arbiter have received m messages, then it begins the decision process based on procedures in Section C and distributes the trustful flow rule to proxy.

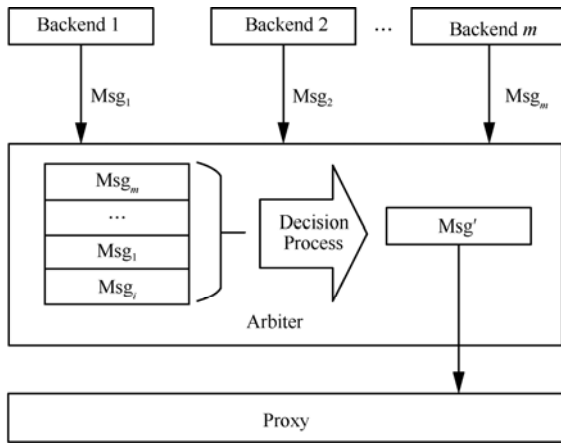


Figure 7 Steps of AISO

② FIFO

This strategy is applied in circumstance where response time can't exceed a threshold. Thus, it's impractical to wait for the arrival of all messages because their arrival time may differ greatly from each other. At the same time, it takes some time to finish the decision process. So FIFO is the preferential choice whose procedures are presented in Figure 8.

When adopting FIFO, the arbiter directly sends the message from the backend which reaches it first regardless of its correctness (Msg_i in the graph). Then the following process is equal to that of AISO till it produces Msg' . And Msg' will be sent to the proxy only when the comparison result between it and the first dispatched rule isn't identical, which means there's something wrong with the previously distributed rule. However, this error won't exert

serious impact on the whole network since it will be rectified in an acceptable time.

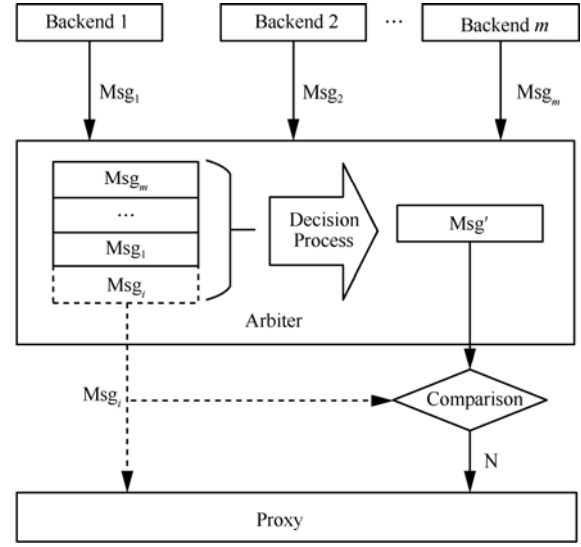


Figure 8 Steps of FIFO

5 Evaluation

In this section, theory analysis and simulation-based experiments are presented to verify the effectiveness of Mcad-SA and coherent method of flow rules we propose.

5.1 Theory analysis of mcad-SA

Actually, it's difficult to estimate precisely the validity of the proposed architecture without understanding the dynamic scheduling strategy it adopts. Thus, we simplify it into static scenes where the set of running controller doesn't vary with time. Given m running controllers, the failure probability of all controllers equals p but they are independent for the controllers are generally diverse. As we know, the condition of collapse of Mcad-SA is over $\left\lceil \frac{m+1}{2} \right\rceil$ controllers are compromised simultaneously. So overall failure probability of Mcad-SA (P_{msa}) can be computed via (1).

$$P_{msa} = \sum_{C_c \geq \left\lceil \frac{m+1}{2} \right\rceil} P^n (1-p)^{m-n} \quad (1)$$

Where $C_c \geq \left\lceil \frac{m+1}{2} \right\rceil$ represents the set of all

occasions with more than $\left\lceil \frac{m+1}{2} \right\rceil$ controllers being compromised simultaneously and n represents the real number of compromised controllers. Based on above assumptions, we can calculate how P_{msa} varies with various combinations of m and p . The simulation result is presented in Figure 9.

From Figure 9, it's obvious to discover that Mcad-SA has significant disadvantage over traditional architectures with merely one controller. What's more, the more controllers it has, the better reliability it can acquire. Besides, the more powerful single controller is, the more distinct security gain it obtains. Taking 3 controllers as the example, when p is 0.25, failure probability of Mcad-SA drops to approximately 0.15, while p decreases to 0.05, P_{msa} reduces sharply to less than 0.01. Thus, improving security of single controller is beneficial to intensify the security of Mcad-SA. In other words, relative techniques about elevating controllers' security can be applied perfectly to Mcad-SA which is an architecture technology in nature. And it's unquestionable that its security can be further strengthened after dynamic is taken into account.

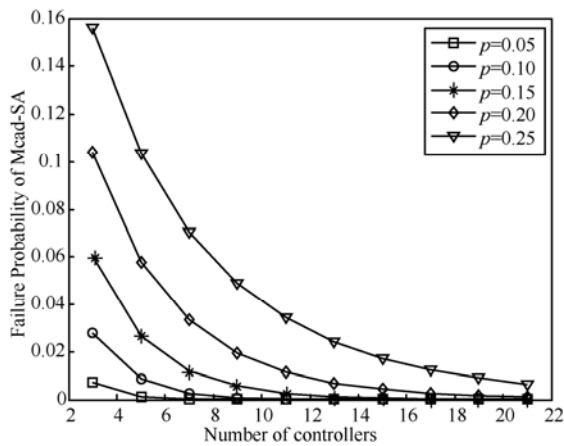


Figure 9 Failure probability of Mcad-SA

5.2 Experimental Results of Mcad-SA

Next, we evaluate the performance of our implementation compared to single common

controllers which Mcad-SA builds on. And three controllers that Mcad-SA adopts are Floodlight, ONOS and ODL^[22]. When we test the average throughput in setting up new entries in the flow table between controllers and switches, we consider controllers operating in a purely reactive way. That means that each switch just has an empty flow table at the beginning, and each new flow request will be forwarded directly to controllers, which will then install flow entries on the flow table corresponding to that request. By the way, the application module running on controllers is learning switch.

Our simulations are implemented in a network simulated in Mininet^[23] which is a network simulation instrument for small-scale prototype. The simulated network we employ is consisted of 131 hosts and 127 switches in a binary tree layout where 4 servers is used for placing three controllers and modules (proxy and arbiter). And the configurations of four servers are listed in Table 2. As the network is a binary tree layout, a single requests generates 13 new flows across 13 switches in the worst case.

1) Results of Mcad-SA's performance

Figure 10 illustrates the measurements we collect under different controllers (the decision strategy that the arbiter takes is AISO). And the measurement metrics we choose are relying on recommended SDN benchmarking metric^[24].

① End-to-End flow setup duration: The average time is required to set up a full path from one host to another.

② Flow setup delay: The time it takes for a single flow entry in the flow table of a single switch to be established.

From Figure 10, it's obvious to discover that Mcad-SA experiences a slow down of around 2x when compared to Floodlight whose performance is the worst of three controllers. This result is not entirely unexpected and it is the increased cost in

Table 2

Notations of fields in the header

Application on Server	IP	CPU	Memory	Operation System
Floodlight	192.168.108.1	2.00 GHz	32 G	Ubuntu16.04
ONOS	192.168.108.2	2.00 GHz	32 G	Ubuntu16.04
ODL	192.168.108.3	2.00 GHz	32 G	Ubuntu16.04
Proxy and arbiter	192.168.108.4	00223-1200 GHz	32 G	Ubuntu16.04

communication complexity and decision process of the arbiter that leads to extra cost of reaching consensus on all flow rules produced by each controller. However, these drops in performance are still acceptable and sufficient to maintain the basic requirement of the Internet. Of course, several methods could be used to mitigate the impact on performance. For example, if the arbiter adopts FIFO, the performance will be a bit lower than that of Floodlight because it's not essential to replace the first distributed rule frequently in most cases. Besides, sample comparison is also a feasible means to guarantee the performance. In this case, the arbiter no longer executes comparison on each rule but at a certain interval. Thus, the performance will be influenced occasionally and it approximates the performance of single controller for most time. In general, Mcad-SA can adjust its deciding strategy to adapt to various situations to satisfy different requirement.

2) Results of rules' coherence in Mcad-SA

Next, we discuss the results of rules' coherence in different occasions in Mcad-SA. In the simulation, we focus on flow_mod rule which is the most important message for switches. To test the effectiveness of Mcad-SA when it faces attackers, we conduct two groups of experiments. One is how

other is how Mcad-SA performs when the rules from controllers are modified.

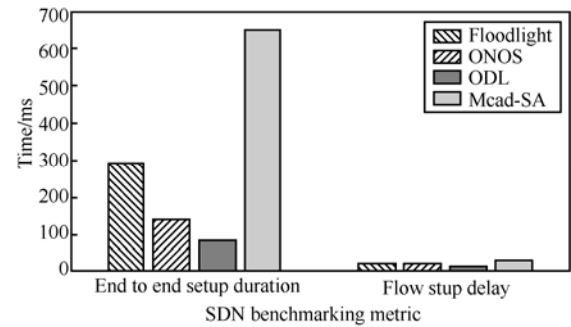


Figure 10 Performance measurement of Mcad-SA

① Fault test of controllers

Case 1 Three controller are operating well

We verify the validity of Mcad-SA via testing the reachability between two hosts. And the test result is presented in Figure 11. Hosts can reach each other without question.

Case 2 One controller is under state of failure

We compromise each controller successively and test the reachability, the test result is presented in Figure 12. As we expected, the system won't be affected when only one controller collapse, which indicates powerful robustness of Mcad-SA.

Case 3 Two controllers are under state of failure

However, the arbiter can't determine whether

```
mininet> h1 ping h2 -c 5
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data:
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=13.7 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.266 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.058 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.082 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.081 ms
--- 10.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 3998ms
```

Figure 11 Reachability between two hosts without any failure

```
mininet> h1 ping h2 -c 5
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data:
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=2.66 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=1.48 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.236 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.081 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.074 ms
--- 10.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4003ms
```

Figure 12 Reachability between two hosts with a controller's failure

Mcad-SA reacts when the controller collapses. The single flow rule it receives is correct or not in this

situation. Therefore, it considers all controllers are malicious and won't transmit the rule to switches. So two hosts are unreachable as Figure 13 shows.

From above results, a conclusion can be made that Mcad-SA is able to tolerate failure of less than $\left\lfloor \frac{m-1}{2} \right\rfloor$ controllers simultaneously if there are m running controllers in the control layer. While in traditional SDN structures, failure of any controller is bound to cause severe influence or damage to the system. Anyway, the cost and difficulty of compromising multiple controllers by attackers is obviously tremendous.

② Test of flow rules modification

Case 1 Flow rules from three controllers are normal

Here, reachability between hosts is also used to test the system's effectiveness. First, we print the flow_mod packets that the arbiter receives from three controllers and present them in Figure 14.

Then we test whether two hosts can be reached or not. As indicated in Figure 15, obviously, the communication between them is without any problem.

Case 2 Flow rules from one controller are modified

In this case, the flow rules of a controller are modified manually, which leads to some changes in the arbiter (Figure 16). The output of the flow rule from Floodlight is altered into 3 while that of the rest is 1. Nevertheless, a right decision is made by the arbiter and reachability between hosts isn't affected (Figure 17).

Case 3 Flow rules from two controllers are modified

In this situation, the arbiter will receive three different flow_mod packets (Figure 18), which is difficult for it to determine which one is right. Thus, it refuses to hand out any flow rule just in case. Of course, packets from one host are unable to reach

```
mininet> h1 ping h2 -c 5
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
From 10.0.0.1 icmp_seq=4 Destination Host Unreachable
From 10.0.0.1 icmp_seq=5 Destination Host Unreachable

--- 10.0.0.2 ping statistics ---
5 packets transmitted, 0 received, +5 errors, 100% packet loss, time 4024ms
```

Figure 13 Reachability between two hosts with two controllers' failures

```
2017-05-16 14:46:47,169 | INFO | OMQ Receive Loop | BackendManager | 159 - core.connectivity - 1.1.0.SNAPSHOT | Message
adds to running request 1 (OpenFlowMessage [Header=MessageHeader
[Version=VERSION_1_4,Type=OPENFLOW,Length=80,TransactionId=34,DatapathId=1],OFMessage=OFFlowAddVer10(xid=75,
match=OFMatchV1Ver10(in port=2, eth dst=ee:09:54:2c:02:74), cookie=0x0010000000000000, id=1, hardTimeout=0, priority=32768,
bufferId=4294967295, flags=[SEND_FLOW_REM], actions=[OFActionOutputVer10(port=flood, maxLen=65535)]))).
2017-05-16 14:46:47,170 | INFO | OMQ Receive Loop | BackendManager | 159 - core.connectivity - 1.1.0.SNAPSHOT | Message
adds to running request 1 (OpenFlowMessage [Header=MessageHeader
[Version=VERSION_1_4,Type=OPENFLOW,Length=80,TransactionId=34,DatapathId=1],OFMessage=OFFlowAddVer10(xid=75,
match=OFMatchV1Ver10(in port=2, eth dst=ee:09:54:2c:02:74), cookie=0x0010000000000000, id=1, hardTimeout=0, priority=32768,
bufferId=4294967295, flags=[SEND_FLOW_REM], actions=[OFActionOutputVer10(port=flood, maxLen=65535)]))).
2017-05-16 14:46:47,171 | INFO | OMQ Receive Loop | BackendManager | 159 - core.connectivity - 1.1.0.SNAPSHOT | Message
adds to running request 1 (OpenFlowMessage [Header=MessageHeader
[Version=VERSION_1_4,Type=OPENFLOW,Length=80,TransactionId=34,DatapathId=1],OFMessage=OFFlowAddVer10(xid=75,
match=OFMatchV1Ver10(in port=2, eth dst=ee:09:54:2c:02:74), cookie=0x0010000000000000, id=1, hardTimeout=0, priority=32768,
bufferId=4294967295, flags=[SEND_FLOW_REM], actions=[OFActionOutputVer10(port=flood, maxLen=65535)]))).
```

Figure 14 Flow_mod packets received by the arbiter in case 1

```
mininet> h1 ping h2 -c 5
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=13.7 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.266 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.058 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.082 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.081 ms

--- 10.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 3998ms
```

Figure 15 Reachability between two hosts without any modification

```

2017-05-16 14:50:33,135 | INFO | OMQ Receive Loop | BackendManager | 159 - core.connectivity - 1.1.0.SNAPSHOT | Message
adds to running request 1 (OpenFlowMessage [Header=MessageHeader
[Version=VERSION 1 4,Type=OPENFLOW,Length=80,moduleId=11],TransactionId=34,DatapathId=1] type=FLOW_MOD) OFMessage=OFFlowAddVer10(xid=75,
match=OFMatchV1Ver10(in port=2, eth dst=ee:05:07:4c:07:4c), cookie=0x0010000000000000, idleTimeout=120, hardTimeout=0, priority=32768,
bufferId=4094167435, outPort=1) flags=[SEND_FLOW_REM], actions=[OFActionOutputVer10(port=flood, maxLen=65535)]))).
2017-05-16 14:50:33,135 | INFO | OMQ Receive Loop | BackendManager | 159 - core.connectivity - 1.1.0.SNAPSHOT | Message
adds to running request 1 (OpenFlowMessage [Header=MessageHeader
[Version=VERSION 1 4,Type=OPENFLOW,Length=80,moduleId=339],TransactionId=34,DatapathId=1] type=FLOW_MOD) OFMessage=OFFlowAddVer10(xid=75,
match=OFMatchV1Ver10(in port=2, eth dst=ee:05:07:4c:07:4c), cookie=0x0010000000000000, idleTimeout=120, hardTimeout=0, priority=32768,
bufferId=4094167435, outPort=1) flags=[SEND_FLOW_REM], actions=[OFActionOutputVer10(port=flood, maxLen=65535)]))).
2017-05-16 14:50:33,135 | INFO | OMQ Receive Loop | BackendManager | 159 - core.connectivity - 1.1.0.SNAPSHOT | Message
adds to running request 1 (OpenFlowMessage [Header=MessageHeader
[Version=VERSION 1 4,Type=OPENFLOW,Length=80,moduleId=0],TransactionId=34,DatapathId=1] type=FLOW_MOD) OFMessage=OFFlowAddVer10(xid=75,
match=OFMatchV1Ver10(in port=2, eth dst=ee:05:07:4c:07:4c), cookie=0x0010000000000000, idleTimeout=120, hardTimeout=0, priority=32768,
bufferId=4094167435, outPort=1) flags=[SEND_FLOW_REM], actions=[OFActionOutputVer10(port=flood, maxLen=65535)]))).

```

Figure 16 Flow_mod packets received by the arbiter in case 2

```

mininet> h1 ping h2 -c 5
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=2.66 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=1.48 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.236 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.081 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.074 ms

--- 10.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4003ms

```

Figure 17 Reachability between two hosts with modification of a controller's flow rules

```

2017-05-16 14:55:43,148 | INFO | OMQ Receive Loop | BackendManager | 159 - core.connectivity - 1.1.0.SNAPSHOT | Message
adds to running request 1 (OpenFlowMessage [Header=MessageHeader
[Version=VERSION 1 4,Type=OPENFLOW,Length=80,moduleId=11],TransactionId=34,DatapathId=1] type=FLOW_MOD) OFMessage=OFFlowAddVer10(xid=75,
match=OFMatchV1Ver10(in port=2, eth dst=ee:05:07:4c:07:4c), cookie=0x0010000000000000, idleTimeout=120, hardTimeout=0, priority=32768,
bufferId=4084369781, outPort=1) flags=[SEND_FLOW_REM], actions=[OFActionOutputVer10(port=flood, maxLen=65535)]))).
2017-05-16 14:55:43,148 | INFO | OMQ Receive Loop | BackendManager | 159 - core.connectivity - 1.1.0.SNAPSHOT | Message
adds to running request 1 (OpenFlowMessage [Header=MessageHeader
[Version=VERSION 1 4,Type=OPENFLOW,Length=80,moduleId=339],TransactionId=34,DatapathId=1] type=FLOW_MOD) OFMessage=OFFlowAddVer10(xid=75,
match=OFMatchV1Ver10(in port=2, eth dst=ee:05:07:4c:07:4c), cookie=0x0010000000000000, idleTimeout=120, hardTimeout=0, priority=32768,
bufferId=4094167435, outPort=1) flags=[SEND_FLOW_REM], actions=[OFActionOutputVer10(port=flood, maxLen=65535)]))).
2017-05-16 14:55:43,148 | INFO | OMQ Receive Loop | BackendManager | 159 - core.connectivity - 1.1.0.SNAPSHOT | Message
adds to running request 1 (OpenFlowMessage [Header=MessageHeader
[Version=VERSION 1 4,Type=OPENFLOW,Length=80,moduleId=0],TransactionId=34,DatapathId=1] type=FLOW_MOD) OFMessage=OFFlowAddVer10(xid=75,
match=OFMatchV1Ver10(in port=2, eth dst=ee:05:07:4c:07:4c), cookie=0x0010000000000000, idleTimeout=120, hardTimeout=0, priority=32768,
bufferId=4094167435, outPort=1) flags=[SEND_FLOW_REM], actions=[OFActionOutputVer10(port=flood, maxLen=65535)]))).

```

Figure 18 Flow_mod packets received by the arbiter in case 3

```

mininet> h1 ping h2 -c 5
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
From 10.0.0.1 icmp_seq=4 Destination Host Unreachable
From 10.0.0.1 icmp_seq=5 Destination Host Unreachable

--- 10.0.0.2 ping statistics ---
5 packets transmitted, 0 received, +5 errors, 100% packet loss, time 4024ms

```

Figure 19 Reachability between two hosts with modification of two controllers' flow rules

the other this moment (Figure 19).

Based on above experimental results, it's clear that Mcad-SA can tolerate the case where flow rules of less than $\left\lfloor \frac{m-1}{2} \right\rfloor$ controllers are modified when it's equipped with m running controllers. Actually, modification of flow rules is a serious issue in SDN with only one controller since it's hard to discover this type of exception. But Mcad-SA can deal with similar situations in an effective means. For attackers, if they desire to conduct successful attacks, they have to control more

than $\left\lceil \frac{m+1}{2} \right\rceil$ controllers simultaneously and ensure the false outputs are the same. And only in this case will they be able to deceive the arbiter and deliver false rules to switches. However, this is nearly an impossible task for running controllers have different vulnerabilities and independent operating environment. Furthermore, it's predictable that the reliability and resilience will be improve significantly when dynamic is introduced into Mcad-SA.

6 Conclusion

Security of SDN has gradually become a vital issue for enabling the network operate reliably. Focused on failures of controllers and flow-tampering attacks, an aware decision-making security architecture with multi-controller is proposed to prevent that kind of attack proactively. Through introducing heterogeneity, redundancy and dynamic into the control plane, the robustness, reliability and resilience of SDN can be significantly strengthened. Furthermore, we conduct profound research on how to ensure the consistency of flow rules from diverse controllers in this architecture. By means of segmentation and grading on different flow rules, we are able to choose the most trustful flow rules to send to underlying switches according to their respective final score (i.e. credibility). Theory analysis and simulation results proves feasibility, reliability and effectiveness of our methods. Actually, the defending concept that combines heterogeneity, dynamism and redundancy of existing techniques, means and components can be applied in various fields other than SDN. In the future, we plan to take dynamic into consideration and implement Mcad-SA completely with the real experimental platform.

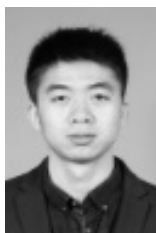
References:

- [1] MCKEOWN N, ANDERSON T, BALAKRISHNAN H, et al. OpenFlow: enabling innovation in campus networks[J]. ACM Sigcomm Computer Communication Review, 2008, 38(2): 69-74.
- [2] MONSANTO C, REICH J, FOSTER N, et al. Composing software-defined networks[C]//The 10th USENIX Conference on Networked Systems Design and Implementation, USENIX Association. 2013: 1-14.
- [3] KREUTZ D, RAMOS F M V, VERISSIMO P. Towards secure and dependable software-defined networks[C]//The Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking. 2013: 55-60.
- [4] CABAJ K, WYTREBOWICZ J, KUKLIŃSKI S, et al. SDN architecture impact on network security[C]//Federated Conference on Computer Science and Information Systems. 2014.
- [5] SCOTT-HAYWARD S, NATARAJAN S, SEZER S. A survey of security in software defined networks[J]. IEEE Communications Surveys & Tutorials, 2015, 18(1): 623-654.
- [6] PORRAS P, SHIN S, YEGNESWARAN V, et al. A security enforcement kernel for OpenFlow networks[C]// 2012: 121-126.
- [7] PORRAS P, CHEUNG S, FONG M, et al. Securing the Software-Defined Network Control Layer[C]//The 2015 Network and Distributed System Security Symposium (NDSS), San Diego, California. 2015. 00223-16
- [8] JAIN S, KUMAR A, MANDAL S, et al. B4: experience with a globally-deployed software defined wan[C]//ACM SIGCOMM 2013 Conference on SIGCOMM. 2013:3-14.
- [9] BERDE P, HART J, HART J, et al. ONOS: towards an open, distributed SDN OS[C]//The Workshop on Hot Topics in Software Defined NETWORKING. 2010:1-6.
- [10] ELDEFRAWY K, KACZMAREK T. Byzantine fault tolerant software-defined networking (SDN) controllers[C]//Computer Software and Applications Conference. 2016: 208-213.
- [11] CELLO M, XU Y, WALID A, et al. BalCon: a distributed elastic SDN control via efficient switch migration[C]//IEEE International Conference on Cloud Engineering. 2017.
- [12] GUO Z, XU Y, CELLO M, et al. JumpFlow: reducing flow table usage in software-defined networks[J]. Computer Networks, 2015, (92): 300-315.
- [13] MCGEER R. A safe, efficient update protocol for openflow networks[C]//The Workshop on Hot Topics in Software Defined Networks. 2012: 61-66.
- [14] SUKAPURAM R, BARUA G. Enhanced algorithms for consistent network updates[C]//Network Function Virtualization and Software Defined Network. 2016: 184-190.
- [15] HUSSEIN A, ELHAJJ I H, CHEHAB A, et al. SDN verification plane for consistency establishment[C]//Computers and Communication. IEEE, 2016:519-524.
- [16] HUA J, GE X, ZHONG S. FOUN: a flow-ordered consistent update mechanism for software-defined networking in adversarial settings[C]//The International Conference on Computer Communications. 2016: 1-9.
- [17] Reitblatt M, Foster N, Rexford J, et al. Consistent updates for software-defined networks: change you can believe in![C]//ACM Workshop on Hot Topics in Networks. 2011:7.
- [18] Ryu SDN Framework[EB/OL]. <http://osrg.github.io/ryu/>.
- [19] FloodLight. "Open SDN controller"[EB/OL]. <http://floodlight.openflowhub.org/>.
- [20] QI C, WU J, HU H, et al. Dynamic-scheduling mechanism of controllers based on security policy in software-defined network[J].

Electronics Letters, 2016, 52(23):1918-1920.

- [21] QI C, WU J, HU H, et al. An intensive security architecture with multi-controller for SDN[C]//Computer Communications Workshops. 2016: 401-402.
- [22] OpenDaylightConsortium[EB/OL]. <http://www.opendaylight.org>.
- [23] LANTZ B, HELLER B, MCKEOWN N. A network in a laptop:rapid prototyping for software-defined networks[C]//ACM Workshop on Hot Topics in Networks. 2010:1-6.
- [24] MACHAT R, BANKS S, CALABRIA F, et al. ISSU benchmarking methocldogy[DB/OL]. <http://tools.ietf.org/html/draft-banks-bmwg-issu-meth-05>.

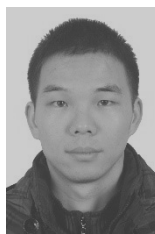
About the authors



LV Yingying (1993-), born in Jiangxi. He is working on his master degree at National Digital Switching System Engineering Technology Research Center. His research interests include cvber securitv and software-defined



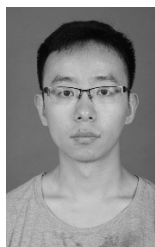
GUO Yunfei (1963-), born in Henan. He is a Ph.D supervisor and professor at National Digital Switching System Engineering Technology Research Center. His main research interests include cloud security, telecommunication network security and cyber security.



QI Chao (1991-), born in Jiangxi. He is working on his Ph.D degree at National Digital Switching System Engineering Technology Research Center. His research interests include cyber security and software-defined network.



WU Qi (1991-), born in Jiangsu. He is working on his Ph.D degree at National Digital Switching System Engineering Technology Research Center. His research interests include cyber security and software-defined network.



WANG Yawen (1990-), born in Henan. He is working on his Ph.D degree at National Digital Switching System Engineering Technology Research Center. His research interests include cloud computing and cvber security.