FOUM: A Flow-Ordered Consistent Update Mechanism for Software-Defined Networking in Adversarial Settings

Jingyu Hua[1,2], Xin Ge[1,2], Sheng Zhong[1,2]
1. State Key Laboratory for Novel Software Technology, Nanjing University, China
2. Department of Computer Science and Technology, Nanjing University, China
Email:huajingyu@nju.edu.cn, mg1333014@smail.nju.edu.cn, zhongsheng@nju.edu.cn

***Abstract*—Due to the asynchronous and distributed nature of the data plane, consistent configuration updating across multiple switches is a challenging issue in Software-Defined Networking (SDN). The existing version-stamping-based mechanism (VSM) could guarantee per-packet consistency, but this mechanism is designed for non-adversarial settings and can be compromised easily by a malicious attacker. In this paper, we propose an efficient flow-ordered update mechanism that aims to provide per-packet consistency in adversarial settings. Our proposal does not need to stamp data packets with the configuration version, and is robust against both the packet-tampering and packet-dropping attacks. It outperforms a naive mechanism that simply patches VSM using digital signatures in three aspects: First, the switches in this mechanism only need to sign and verify a single control packet, which significantly improves the packet processing time. Second, it avoids keeping both old and new policies on switches during the update, and thus achieves better space efficiency. Third, it reduces the time delay for new policies to come into force. We evaluate our mechanism on a self-constructed SDN testbed and the results demonstrate high efficiency.**

***Index Terms*—Consistent Update, Software Defined Networking, Network Security**

I. Introduction

Software Defined Networking (SDN) decouples the control plane, which makes decisions about how traffic is forwarded, from the underly switches, which forward traffic (the data plane). Network administrators are allowed to run their programs on a centralized controller to remotely configure the packet-handling policies on switches at a high level of abstraction. Compared with the traditional network architecture, SDN makes it much easier for administrators to make frequent updates to the network configuration.

A configuration update usually adds, modifies or removes some policies across multiple switches. As the switches are distributed, we cannot guarantee that these updates reach all the switches at the same time. This leads to a sequence of intermediate configurations before reaching the completely new one. Consequently, a packet entering the network during this period may be processed by a combination of the old and new policies. Such inconsistency may bring serious consequences such as forwarding loops, packet losses and access control violations [4, 6, 7, 12].

This work was supported in part by NSFC-61321491, NSFC-61425024, NSFC-61300235, NSFC-61402223, and Jiangsu Province Double Innovation Talent Program.

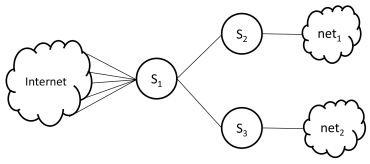


Fig. 1: A firewall example.

Consider, for example, the network in Fig. 1. There are three switches in this network: an ingress switch S_1, and two internal switches S_2 and S_3 that are connected to the internal sub-nets net_1 and net_2, respectively. Packets arriving at S_1 are forwarded to S_2 or S_3 based on their destination addresses. Initially, only the firewall function of S_1 is turned on. However, later, as the administrator finds that S_1 is overloaded by the packet filtering, he issues an update to offload this task to S_2 and S_3. Due to the distributed nature of the switches, we cannot predict the order in which the switches receive and apply the new policies. If S_1 is updated before S_2 or S_3, we may get an intermediate configuration where all of the switches turn off their firewall functions, which may let in attack traffic. A valid update plan should guarantee that S_2 and S_3 are updated before S_1.

Finding a good update order requires complicated reasoning about implicity dependencies between policies or an exhaustive search among all possible orders [12, 13], both of which are extremely complicated in a large-scale network. In certain cases, it may be even impossible to find a faultless order [13]. To bypass these difficulties, Reitblatt et al. [13] propose a *version-stamping based mechanism* (VSM), which stamps the packets with the configuration version at ingress switches and requires internal switches to process packets using the policies with the appropriate version numbers. VSM is demonstrated to provide the per-packet consistency, i.e., each packet is processed either using the old policies or using the new ones, but never using their combinations.

While VSM is a an elegant solution to the policy update problem, it is designed for *non-adversarial* settings only. In a real network, an attacker may compromise and intercept switch-switch communication using various attacks [1, 3, 5, 10, 14]. In this case, VSM may be easily compromised. Specifically, an attacker may tamper with the version numbers stored in the packets, so that per-packet consistency is no longer guaranteed. We can certainly defend against such attacks by

making ingress switches sign and verify the version numbers. (Hereafter, we refer to this patched mechanism as *Secure Version-Stamping Based Mechanism* (SVSM)). Nevertheless, it is too heavyweight to generate a digital signature for each data packet and verify the signature on every internal switch.

In this paper, we aim to propose an efficient policy update mechanism that can provide per-packet consistency in adversarial settings. Compared with SVSM, **our mechanism has the following advantages**:

(1) Although our mechanism also uses digital signature, it requires a single signature for each update. Hence, packet processing under our mechanism is much faster.

(2) Under our mechanism, switches do not store both old and new policies simultaneously during the update. Hence, the higher space efficiency of our mechanism is also better.

(3) Our mechanism allows new policies to take effect faster.

More specifically, our contributions in this paper can be summarized as follows:

- We are the first to study the problem of consistent network update in adversarial SDN environments.
- Considering that real network updates involving multiple paths can be logically divided into multiple single-path updates. We propose an efficient flow-ordered update mechanism, which can provide per-packet consistency in adversary settings, for individual paths. We demonstrate that it imposes smaller side effects on the data throughput than SVSM and has the ability to pinpoint problematic links.
- We then address the challenge to automatically partition the mixed updates from administrators into a set of single-path updates that do not interfere with each other. We also extend FOUM to provide per-session consistency (i.e., per-flow consistency in [4]), which is stronger than per-packet consistency.
- We conduct simulations on a SDN testbed to evaluate the performance of our mechanism. The results verify that our mechanism outperforms SVSM both in the update speed and in the data throughput during the update.

II. Preliminaries

In this section, we first describe the network model for the problem of consistent updates in SDN, and then present the adversary model and the design goal in this paper.

A. Network Model

Table I defines the primary elements in our network model. We assume that there are n switches $\{s_1, \ldots, s_n\}$ and one controller C in the network. A packet Pt is a 2-tuple $(header, payload)$, where $header$ is the same with the IP header.

In our model, we assume every switch establishes at least two distinct channels to the controller for tolerating the link outages. In addition, each switch maintains a flow table FT, which is used to perform packet matching and forwarding. FT consists of a set of ordered flow entries; every entry FE is a 2-tuple $(matchfields, actions)$. The packets are matched against the match fields of flow table entries and the actions of the first matching entry will be executed. The possible actions include modifying packets and putting packets to the output queue of a specific port. According to the specification of OpenFlow [9], there can be multiple flow tables on the same switch. We assume there is only one flow table on each switch for simplicity.

TABLE I: Network Model

element name	description
switchID	$SID \in \{S_1, S_2, \ldots, S_n\}$
path	$P = [S_{k1} \rightarrow S_{k2} \rightarrow \cdots \rightarrow S_{kn}]$
packet	$Pt = (header, payload)$
port	$P \in \{p1, p2, \ldots, pn\}$
match field	$MF(Pt) = (Pt.header, Pt, SID)$
flow entry	$FE = (match\ fields, actions)$
flow table	$FT = \{FE_1, FE_2, \ldots, FE_n\}$
configuration	$Con = \{(SID_1, FT_1), (SID_2, FT_2), \ldots, (SID_n, FT_n)\}$
update entry	$UE : FE \rightarrow FE'$
update	$U = \{(S_{k_i}, UE_{t_i}) \mid i = 1, 2, \cdots, m\}$

At the runtime, the controller can add, modify and delete flow entries on each switch through the controller-switch channels. We assume that every update U is defined to be a set $\{(S_{k_i}, UE(FE_{t_i}, FE'_{t_i})) \mid i = 1, 2, \cdots, m\}$, where each elemental update $UE(FE, FE')$ is a function defined for a specific switch S_i:

As not all the switches are directly connected to the controller, there must be varied delays between the controller and switches. This indicates that, due to the distributed nature of the data plane, it is unlikely for elemental updates to reach the target switches and take effects at the same time. As a result, the same packet or packets in the same TCP session may be processed by inconsistent flow entries on different switches. Let $Old(U) = \{(S_{k_i}, FE_{t_i}) \mid i = 1, 2, \cdots, m\}$ and $New(U) = \{(S_{k_i}, FE'_{t_i}) \mid i = 1, 2, \cdots, m\}$ denote the sets of old and new flow entries that U involves, respectively. A consistent update should guarantee that the same packet or the packets in the same TCP session should be processed by only an individual configuration: either rules in $Old(U)$ or those in $New(U)$, but not a mixture of the two. We formally define two kinds of update consistency introduced in [12]:

Definition 1 (Per-packet consistency)**.** Let $Real(Pt) = \{(S_{j_i}, FE_{p_i}) \mid i = 1, 2, \cdots, w\}$ denote the list of switches that a packet Pt takes through the network, as well as the matching flow entries on these switches. An update U from the controller is per-packet consistent i.f.f. before, during and after the update process, $\forall\ Pt$ that traversed the network, $\nexists e_1, e_2 \in Real(Pt)$ that $e_1 \in Old(U) \wedge e_2 \in new(U)$.

Definition 2 (Per-session consistency)**.** Let $Real(Se) = \bigcup_{Pt \in Session} Real(Pt)$ denote the list of switches that the packets of a TCP session Se take through the network, and the matching flow entries on these switches. An update U from the controller is per-session consistent i.f.f. before, during and after the update process, $\forall\ S_n$, $\nexists e_1, e_2 \in Real(Sn)$ that $e_1 \in Old(U) \wedge e_2 \in new(U)$.

In this paper, we will focus on the per-packet consistency.

Nevertheless, we also discuss how to extend our proposal to satisfy the per-session consistency.

B. Adversary Model

We assume that the centralized controller is always trusted and the distributed switches are benign at the beginning. An adversary may launch physical attacks or exploit switch vulnerabilities to intercept the communication between switches. Once a link is compromised and hijacked, we assume that the adversary may do two evil things against the flying packets:

- manipulates the packets or packet fields introduced by the update mechanism;
- directly drops packets introduced by the update mechanism.

Note that this paper does not consider attacks targeting update-independent packets or fields. That is, we only focus on the attacks unique to the update mechanism. For instance, in VSM, we will consider the attacks tampering with the tag field, which is introduced by VSM, but will not consider attacks tampering with other standard IP-header fields such as the source or destination IP addresses. Although these attacks may also affect the consistency of updates, they usually require a more general defense mechanism, which is beyond the scope of this paper.

C. Our Goal

This paper aims to design a robust update mechanism that can guarantee per-packet consistency confronting the above adversary model. To be specific, our design goal consists of the following three aspects:

(1) **Consistency**: The update process can provide per-packet consistency facing our adversary model.

(2) **Integrity & Authenticity**: The switches can guarantee that every update they apply to their flow tables is correct and really issued by the controller. In other words, the switch should have the ability to verify the integrity of an update request from the controller.

(3) **Accountability**: The controller has the ability to detect whether an update U is correctly deployed. If something wrong happens, the controller can quickly locate the abnormal links or the malicious switch.

(4) **Efficiency**: The efficiency of the update should be as high as possible.

III. A naive mechanism by patching the existing version-stamping approach

Reitblatt et al. [12] proposes a version-stamping update mechanism that can provide per-packet consistency in a network without any adversaries.

It is easy to prove that this mechanism is per-packet consistent under the assumption that there are no adversaries. However, if there does exist an adversary that launch attacks described in the last section, this conclusion no longer holds. Specifically, when the version of a packet is manipulated at a switch, it must be processed by inconsistent policies before and after this switch.

PATCH : The above inconsistent is due to the malicious modification of the version number of the packets. Hence, we can easily propose a patch based on digital signatures. Specifically, when stamping the packets entering the network with version number in the $header$ field, the ingress switch signs the $header$ of the packet. We use the function $Sign\,(header)$ to perform this step, and the generated signature is denoted by sig. We put sig in the head of $payload$. In addition, to prevent replay attacks, we include a timestamp in the $header$ (in the $options$ field). The internal switches will verify sig of every received packet before processing them. In addition, the communication between the controller and any switch should also be protected by digital signatures to guarantee the integrity of the update messages.

Once the signature verification of a packet fails on a switch, this switch will drop the packet and then raise an alarm to the controller from a path without the incoming link of the packet. We denote the trace of the packet Pt in the network by $T = \{S_1, \cdots, S_{i-1}, S_i, \cdots\}$ and the link between S_{j-1} and S_j by L_j. If S_k is the first switch informs the controller that the packet Pt it receives has been manipulated, the controller can conclude that some attack or error has occurred on L_k if all the switches are trusted.

For simplicity, we name VSM applying the above patch *Secure VSM (SVSM)*. It is easy to prove that SVSM can guarantee per-packet consistency confronting our adversary model. Nevertheless, SVSM requires to generate a digital signature for every data packet entering the network and then verify them at each internal switch, which is extremely time consuming that may significantly hurt the network throughput.

IV. Our Proposal: Flow-Ordered Update Mechanism

Although SVSM can guarantee the consistency by preventing the link or the switch from tampering with the version number of the packet, it will bring unaffordable side effects to the network performance. In this section, we present a novel update mechanism that can be efficiently enhanced to defend against attacks. We first present its non-adversary version, and then present how to efficiently protect it from attacks and other abnormalities which will make consistency fail.

A. Mechanism Overview

In this subsection, we will put forward the consistent update mechanism in a non-adversarial settings as below:

(1)There is no attacker in the network that can manipulate and modify the packets or drop the packets deliberately.

(2)We assume all the packets will not be dropped for other reason, such as the link fails to work because of the broken cable.

We will present the improved solution which can deal with the abnormal situation that violates one or more of the two assumptions above in the next subsection.

Real-world network updates usually involve multiple paths. However, as we will introduce in the next section, a multi-

path update can be logically divided into a set of single-path updates. So our basic mechanism considers single paths.

Before we present our proposal, we first give the definition of *Path*.

Definition 3 (Path)**.** A path $P = S_{k_1} \rightarrow S_{k_2} \rightarrow \cdots \rightarrow S_{k_n}$ refers to a non-branched and ordered sequence of switches that the packets of one or several flows traverse as they are forwarded through the network.

A single-path update then aims to reconfigure the flow entries matching flows on a specific path.

Example 1. *In Fig. 2, there is a flow F flying along the path $P = S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_n$. Those updates that only change the policies matching F on $S_1, S_2, \cdots, S_n$ are typical single-path updates.*

In Example 1, we assume that at some time the packets waiting to be processed in the import queue of the ingress switch S_1 are $(Pt_{k-1}, Pt_u, Pt_k, Pt_{k+1})$. And these packets will be forwarded along the path P.

Suppose that some time later, the administrator hopes to update the switch policies for F. For this purpose, we make the central controller packs all the update messages for switches on P into an individual control packet Pt_u and inserts it in the front of the import queue of the ingress switch. So, at time t_2, the import queue of S_1 after the insertion becomes (Pt_u, Pt_k, Pt_{k+1}). Each switch can identify Pt_u by a field named $flag$ in the $header$. The payload of Pt_u is a 2-tuple $(update\_list, sig)$, where $update\_list$ is a list of update messages and sig is the digital signature of $update\_list$. Each update message in $update\_list$ targets for a unique switch and is defined to be a triple $(S_i, U(S_i), S_{next})$. Here, S_i represents the target switch of the message, $U(S_i)$ is a set composed of all the update entries for S_i, and S_{next} is the next-hop switch of path P. This control packet will travel along the target path the same as ordinary data packets in F.

The control packet Pt_u is in front of Pt_k and Pt_{k+1} while behind Pt_{k-1} in the import queue of S_1. After being processed by S_2, Pt_u is still in front of Pt_k and Pt_{k+1} while behind Pt_{k-1} in the output queue. No matter the order of the front packets($\cdots\ Pt_{k-1}$) is or the order of the behind packets(Pt_k, $Pt_{k+1}\ \cdots$) is, Pt_u has the ordering attribute that the packets in front of it will still in front of it and the packets behind it will still behind it after they are processed by a switch.

When the switch S_i dequeues a packet from the ingress queue, S_i first check the $flag$ field in the header. If this is a data packet, S_i processes it with the current policies in its flow table. If this is a control packet, S_i extracts the update message targeting himself and directly deploys all the update entries within this message. Note that, different from VSM, the deployment here will replace the old policies with new ones instead of keeping both of them. By doing so, all the subsequent data packets after Pt_u in the ingress queue of S_i will be processed with new policies on S_i. For instance, in Example 1, if the packets waiting in the import queue of S_2 is $(Pt_{k-2}, Pt_{k-1}, Pt_u, Pt_k, \cdots)$, it indicates that Pt_{k-2} and Pt_{k-1} will be processed with old policies while all the remained packets will be processed with new policies. In the end, this control packet will be forwarded to the next-hop switch S_{next} in this path. The whole process is presented in Fig. 2. As this mechanism updates the switches in an order determined by the flow, we name it ***Flow-Ordered Update Mechanism (FOUM)***.

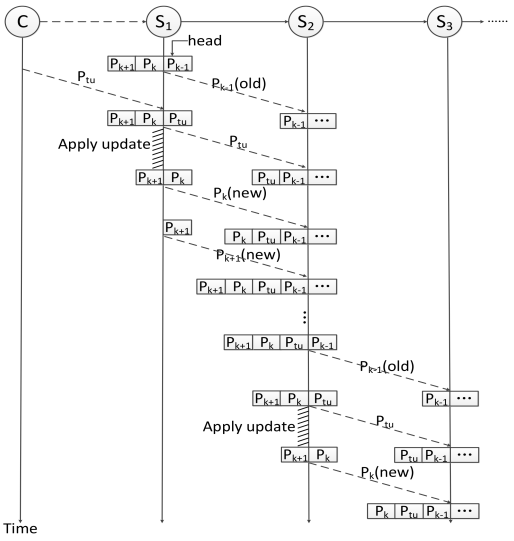


Fig. 2: Illustration of FOUM

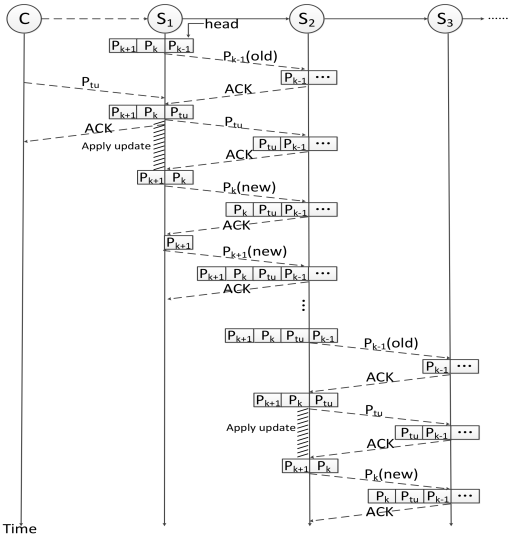


Fig. 3: Next-hop ACK method for defending against packet dropping attacks

keep the ordering attribute of the control packet: Let's take the example $(Pt_{k-1}, Pt_k, Pt_u, Pt_{k+1}, Pt_{k+2}\cdots)$ on switch S_i. There are several concurrent processes to deal with the queued packets in a switch. As we described before, Pt_u is being processed in one of the concurrent processes before Pt_{k+1} and Pt_{k+2}. And if packets are more complicated to process so they will need much more time. If the control packet Pt_u is much more time consuming on S_i to process than regular data packets Pt_{k+1} and Pt_{k+2}. And this will lead to the result that Pt_{k+1} and Pt_{k+2} may be sent to

the output port before Pt_u. Thus will lead to the attribute of the ordering-packet Pt_u invalid. Pt_{k+1} and Pt_{k+2} that saw the new configuration before S_i will jump ahead of the control packet and continue through the network using the old configuration after S_i.

Our solution to this problem is to increase the priority of control packet with respect to the packets behind the control packet. Pt_u will have the higher priority compared with the packets such as Pt_{k+1} and Pt_{k+2} which are behind it in the import queue. Once arrives a control packet, the switch will wait a moment until all the packets in front of the control packet Pt_u have arrived at the import packet. When the control packet Pt_u is processed by the switch, other packets in the import queue will wait until the control packet Pt_u has been processed and arrived at the output queue. This will guarantee the ordering attribute of the control packet.

It is easy to prove that this FOUM can guarantee per-packet consistency for single-flow updates in non-adversarial settings. Due to the space limit, we omit the proof here. Our attack countermeasures in the next section will extend this conclusion to a malicious network.

Compared with VSM, this proposal has the following *advantages*:

(1) During the update, each switch keeps either the old or the new policies, but never both of the two−not requiring to double the rule space as VSM.

(2) New policies start to be used to handle packets as soon as the update packet reaches the ingress switch. SVSM should, however, wait until all the internal switches have received their updates. Therefore, new policies in our mechanism can take effects faster.

B. Countermeasures against attacks

The basic mechanism is high-efficient in both time and space. However, it does not consider the adversary model that we described in Sec. II. This subsection discusses how to protect FOUM and enable it to provide per-packet consistency confronting attacks.

According to our adversary model, in FOUM the compromised links or the fake links[3] may launch the following attacks:

- manipulate the control packet, including the $flag$ field and the update messages;
- completely drop the control packet.

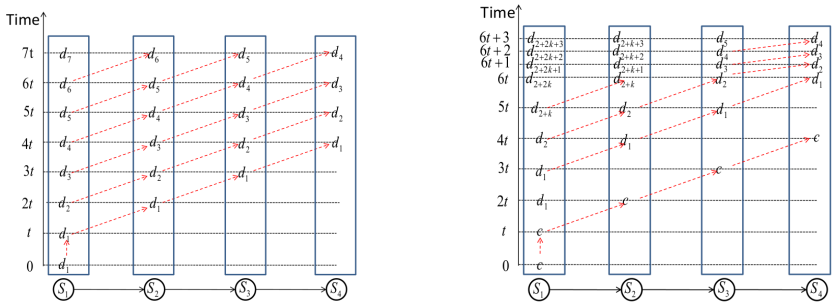
The manipulation of update messages can be easily prevented because we include a signature of the update messages in every control packet. Since it only requires to verify signatures of control packet, the throughput of the network will not be affected as seriously as in SVSM.

When the $flag$ of a control packet is changed, the subsequent switches will regard it as a normal data packet, so will miss the update messages. This may breach the per-packet consistency because the prior switches have been equipped with new policies. We find that this attack has the same effect with those to directly drop control packets. We now present a unified countermeasure against these two kinds of attacks.

Without loss of generality, we assume that it is the link L between switches S_i and S_{i+1} that has been compromised. Both S_i and S_{i+1} are benign. In this case, Whether L manipulates the $flag$ field of a control packet or directly drop this control packet, S_{i+1} can no longer receive its update messages. As in Fig. 3, we thereby propose a *next-hop ack method* based on this observation to defend against these two attacks. In particular, when a switch S_{rcv} receives a control packet from S_{snd}, we make it reply with an ack message before inserting this packet into the import queue. S_{snd} will wait for this ack message for a short period of time, during which it pauses to send any packets to S_{rcv}. In this case, if the import queue is filled up, we will use the method in [7] to send the packets to the controller temporally. If the ack message does not come until the time out, S_{snd} will send an alarm as well as subsequent packets of the current flow to the controller. Certainly, to prevent a malicious link from forging an ack message, S_{rcv} should sign its ack message. After applying this countermeasure, the controller in the above example will receive a report from S_i saying that it does not receive the acknowledgement from S_{i+1}. If the administrator knows that both S_i and S_{i+1} are legitimate, he can conclude that some error or attack has occurred on L.

This next-hop ack method will certainly slow down the packet transmission. However, since we only run it for the control packet, this side effect is not serious. Both our theoretical analysis (See Sec. IV-C4) and experiments show that FOUM applying this patch imposes smaller side effects on the data throughput than SVSM.

To make our mechanism as secure as possible, we further consider an enhanced adversary model where some switches are compromised and become dishonest. Suppose that S_{i-1}, S_i, S_{i+1} are three continuous switches on the path of a flow, and S_i has been compromised and is dishonest. In this case, if S_i manipulates the $flag$ field of a control packet Pt_u from S_{i-1} or directly drop Pt_u, the next-hop ack method cannot capture it because S_i normally replies S_{i-1} with a legal ack message and does not report itself to the controller. Nevertheless, we find that under these two attacks, the next next hop, S_{i+1}, cannot receive or notice receiving the control packet (We assume that malicious switches do not collude with each other). We can thus use a *next-next-hop ack method* to handle these attacks. In other words, we make S_{i-1} wait for a signed ack message Pt_{ack} from S_{i+1} (the next next hop) rather than S_i (the next hop). If S_{i-1} does not receive the ack message from S_{i+1} within a specified time slot, it will send an alarm as well as the following packets of the flow to the controller. However, when the controller receives an alarm from S_{i-1}, the situation becomes a little bit complex because each of the three is under suspicion: S_{i-1} may have triggered a false alarm, S_i may have dropped the control packet, and S_{i+1} may have intentionally refused to acknowledge S_{i-1}. To more precisely pinpoint the problem, the administrator has to rely on addition information from other sources. Otherwise, he has to isolate all of the three switches for security.



(a) SVSM (b) FOUM

Fig. 4: Illustration of packets transmission along the path of a flow during update

C. Analysis

We will analyze FOUM from the four aspects that we mentioned in the end of Sec. II.

1) Per-Packet consistency: In Sec. IV-A, we have proved that FOUM can provide per-packet consistency in single-flow updates in the absence of attacks. We now analyze whether improved FOUM can keep this property confronting our adversary model.

Suppose that, in a specific flow, all the switches and links before and including S_i are legitimate, and a control packet Pt_u now arrives at S_i. In this case, every data packet after Pt_u has been processed with new policies on switches prior to S_i. According to FOUM, S_i will forward Pt_u to S_{i+1} after checking its integrity. If the next link between S_i and S_j is compromised and then manipulates the $flag$ field or directly drops Pt_u, S_i cannot receive the acknowledge from S_{i+1}. As a result, S_i can detect this attack and will not forward subsequent data packets to S_{i+1}, which prevents these data packets from being inconsistently processed with old policies on switches after S_i. So, we come to the following conclusion.

Theorem 1. *Improved FOUM can guarantee per-packet consistency confronting the adversary model in Sec. II.*

2) Integrity & Authenticity: As every control packet is signed by the controller, any switch receives it can easily verify the integrity and authenticity of the update messages inside it. So, improved FOUM can well guarantee the integrity and authenticity of the applied updates.

3) Accountability: According to our analysis in Sec. IV-B, the next-hop ack method can help the administrator precisely trace the problematic links. Even in the enhanced adversary model, the next-next-hop ack method can pinpoint one or several suspicious switches. So, improved FOUM satisfies the accountability requirement.

4) Efficiency: Comparing the efficiency of SVSM and improved FOUM, we have the following theorem:

Theorem 2. *Improved FOUM (applying the next-hop ack protection) outperforms SVSM in terms of flow throughput.*

Proof: Suppose that the path of the flow to be updated contains n switches, and it takes one time unit for a packet to be transmitted between two adjacent switches (including the time for matching flow entries), and the time costs for one time of signature generation and verification are both t. We also assume that the SVSM-based update for this flow lasts for T time units. Then, according to our sample analysis in Fig. 4a, we can find that, in SVSM, it takes $(n+1)t$ time unites for the first data packet to reach the export port of the last switch S_n. The arrival interval of the following packets is t. Thus, the total number of packets that traverse the network in this flow during T is $M = \frac{T-nt}{t} + 1$.

For FOUM applying the next-hop ack protection as in Fig. 4b, it takes the first data packet $(n+2)t$ time unites to arrive at the export port of S_n. This time is longer than that of VSM. However, the arrival interval of the later packets is only one time unit, which is much shorter than in SVSM. The total number of packets traversing the network in this flow during T is $M = T - (n+2)t + 1$.

Based on the above two equations, we can easily derive that FOUM outperforms SVSM in terms of flow throughput when $T > nt + 2t^2/(t-1)$. Recall that in the third step of VSM, switches have to wait a time period to let all the packets with old-version tags leave the network. This period should be much longer than nt, which is the minimum time for a data packet to pass through the path of the flow, considering the existence of unexpected delays and congestions. In addition, as the internal switches cannot predict the exact time for the ingress switch to stamp the incoming packets with new version numbers, they have to verify the version tags of received packets at least before the second step. As in this step, the ingress switch has to verify the integrity of the update message from the controller, the time period for this step is at least similar with $2t^2/(t-1)$. Therefore, the above inequity is satisfied in the real world, and we can come to the conclusion in this theorem. ■

V. Further Extensions of FOUM

This section considers two extensions of FOUM. We first propose an algorithm for automatically transforming multi-path updates into single-path updates, which enables FOUM to deal with multi-path updates. We then discuss how to extend FOUM to provide inter-session consistency (i.e., per-

flow consistency in [4]), which is stronger than per-packet consistency.

A. To deal with multi-path updates

Many real-world updates involve more than one path. Just consider a common update to change a flow's route. It has to first set up a new path and then release the old one, which obviously involve two paths. We thus need an algorithm to transform multi-path updates into a set of single-path updates.

In our design, this task is performed on the controller, and transparent to user programs on the application level. We assume that an update U from the user program is a mixed set of elemental updates for different switches, $U = \{(S_{k_i}, UE(FE_{t_i}, FE'_{t_i}))|i = 1, 2, \cdots, m\}$. Our goal is to divide these updates into subsets that each is composed of updates targeting switches on the same path.

For this purpose, we make the controller maintain t-wo tables in their configuration database. The first one is $ActPath[ID, Path]$, which records the paths of active flows, and the other one is $FE\_Table[ID, SID, FE, PathID]$, which records the relation between every flow entry and the path of the flow that it matches. Every time when a controller receives a request from a switch to construct a flow, it will add the path assigned to this flow to $ActPath$, and insert the relations between this path and the corresponding flow entries in FE_Table. Then, the proposed algorithm is presented in Algorithem 1.

Algorithm 1: Transforming a multi-path update into single-path updates

Inputs : A multi-path update, U;
Output: A set of single-path updates, Out

```
1  begin
2      foreach u ∈ U do
3          P = FindPath(FE_Table, u.FE, ActPath);
4          if Out[P.ID] ==NULL then
5              Out[P.ID].update_list[u.S] ← u;
6          else
7              Create a control packet Pt_u;
8              Initialize(Pt_u, P);
9              Out[P.ID] = Pt_u;
10     foreach Pt_u ∈ Out do
11         Sign(Pt_u)
12     return Out
```

After running this algorithm, we will obtain a set of control packets, each of which corresponds to a single path. The controller then inserts all of them in the head of the import queue of the ingress switch. By doing so, sub-updates for distinct paths will be performed independently. After all these sub-updates finish, the whole update completes. Nevertheless, if we do not consider the dependencies between paths, the obtained single-path updates may interfere with each other, and thus breach the per-packet consistency when several flows converge at a switch.

Example 2. *Consider the network in Fig. 5. We assume that there are two flows $flow_1$ and $flow_2$ along the path $S_1 \rightarrow S_2 \rightarrow S_4 \rightarrow S_5$ and $S_1 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5$, respectively. The packets in $flow_1$ and $flow_2$ are destined for* 10.0.0.0 *and* 10.0.0.1*, respectively. The old policy on S_4, denoted by $FE_o = (10.0.0.*, Action_o)$ can match both $flow_1$ and $flow_2$. Now, the administrator wants to replace FE_o with a new policy $FE_n = (10.0.0.*, Action_n)$, where $Action_n \neq Action_o$. According to the above solution, this update message will be included into the control packets (denoted by Pt_{u1} and Pt_{u2}) for both $flow_1$ and $flow_2$. However, it is usually unlikely for Pt_{u1} and Pt_{u2} to arrive at S_4 at the same time. Without loss of generality, we assume that Pt_{u1} of $flow_1$ arrives earlier. In this case, there may exist some data packets of $flow_2$ that arrive at S_4 later than Pt_{u1} but earlier than Pt_{u2}. These packets are processed inconsistent policies before and after S_4.*

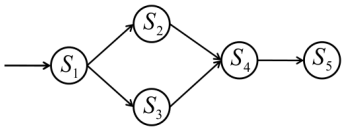


Fig. 5: A network for illustrating multi-path updates

We propose a patch to solve this problem, which is composed of two major parts:

First, when the controller creates a control packet for a sub-update, we make it check whether this path P intersects with any others. If so, the controller should further determine whether the new policies defined for converging switches (i.e., S_4 and S_5 in Example 2) in this sub-update satisfy two conditions: (1) matching packets in other paths, but (2) inconsistent with their present policies. For such policies, we make the controller add a flow constraint such that they can only match flows on P after being deployed. For instance, in Example 2, the new policy $(10.0.0.*, Action_n)$ for S_4 satisfies the above two conditions in both flows. So, we add a flow constraint, and replace it with $(10.0.0.0, Action_n)$ and $(10.0.0.1, Action_n)$ in Pt_{u1} and Pt_{u2}, respectively. By doing so, no matter which of the two control packets arrive at S_4 earlier, the new policy can only affect the subsequent packets in flows on its own path before the other arrives.

Second, with the above measure, one policy defined for a converging switch may be split into multiple, each of which is defined for an individual path. It is obviously space inefficient to store them separately in the flow table. So, when a converging switch receives multiple control packets from different paths, it should merge those policies that can be merged. For instance, the two policies $(10.0.0.0, Action_n)$ and $(10.0.0.1, Action_n)$ in our example can be finally merged to $(10.0.0.*, Action_n)$ again after S_4 being updated.

B. Per-session consistency

As we analyze above, FOUM satisfies per-packet consistency. We now discuss how to extend to provide per-session consistency. The same as the second per-session consistent mechanism in [12], we exploit the wildcard *clone* feature of the DevoFlow extension [2] of OpenFlow to fulfill this goal.

When a packet matches a wildcard clone rule, a DevoFlow switch creates a new rule that matches this packet's header fields exactly. All the subsequent packets in the *micro-flow* defined by this packet's header fields are processed with this

new rule. This actually helps the switch maintain a record of all the active flows in its flow table. We can combine FOUM with this feature to implement a per-session consistent mechanism. Specifically, we first turn on the clone feature of every switch before update. Second, during the update, we use FOUM to update the clone rules on each switch. By doing so, packets in the existing sessions are still handled by the old clone rules, while the packets in the new sessions after the control packet will be processed with new clone rules. This satisfies the per-session consistency.

VI. Evaluation

This section discusses our experiments to evaluate FOUM on a self-constructed SDN testbed. The results demonstrate its high efficiency.

A. Experiment Setup

For the evaluation purpose, we construct a small-scale testbed, which is composed of six servers simulating five switches and one controller, respectively. We use servers to simulate switches because our mechanism (VSM, too) requires to revise the forwarding programs of switches, which are not supported by most commodity switches. We abstract away all the hosts connected to the switches. To make up for them, we develop a packet generator on the ingress switch to produce packets when needed.

In our experiments, we consider the following three scenarios:

(1) Single-path updates: in this simplest scenario, we assume that the five switches form the path of a data flow. The controller wants to add a new policy (i.e., add a source-port constraint) on each node for this flow. We vary the length of the path to evaluate the effects of the flow-length on the update efficiency.

(2) Routing-handoff updates: Consider the network in Fig. 5. We assume that, originally, there is a flow along the path $S_1 \rightarrow S_2 \rightarrow S_4 \rightarrow S_5$. However, at some time, as the administrator wants to use S_2 for other purpose, he issues an update to change the route of this flow to $S_1 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5$. **This update involves two paths, which will not interfere with each other**.

(3) Load-balance updates: Still consider the network in Fig. 5. We assume, originally, there is only one flow along the path $S_1 \rightarrow S_2 \rightarrow S_4 \rightarrow S_5$. The packets in this flow are destined for 10.0.0.0 or 10.0.0.1. S_4 is enabled the firewall function to examine every packet passing through it. However, due to the load increasing, the packet filtering task of S_4 becomes the bottleneck of this flow. The administrator then issues command to offload the flow destined for 10.0.0.1 to the new path $S_1 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5$. In addition, the packet examination is also offloaded to S_2 and S_3, respectively. **This update involves two paths that may interfere with each other**.

In our experiments, we introduce two metrics to evaluate the performance of FOUM and the other two baselines (i.e., VSM and SVSM):

- $\varepsilon = \frac{TP_{during}}{TP_{before}}$, where TP_{before} and TP_{during} are the network throughput before and during the update process. A greater ψ indicates that the side effect on data throughput is smaller, i.e., the update is more efficient.
- T_e, the time elapse from the controller starts the update to the first packet using new policies leaves the network. A smaller T_e indicates that the new policies take effects faster, i.e., the update is more efficient.

For simplicity, all the business logics of the servers are performed in the application layer. This is different from real-world switches, where most business logics are performed in the network layer. Nevertheless, we think our experiments are still meaningful because (1) the time costs for the data transmissions between these two layers can be regarded additional link delays between switches, and (2) we compare FOUM with VSM and SVSM on the same testbed.

The size of each packet generated by the packet generator is 1352B. We set a queue for each switch to keep the packets waiting to be processed and the size of the queue is 20000B. We use a RSA-based signature scheme to generate the required signatures. The key size are 256 bits.

B. Experimental Results

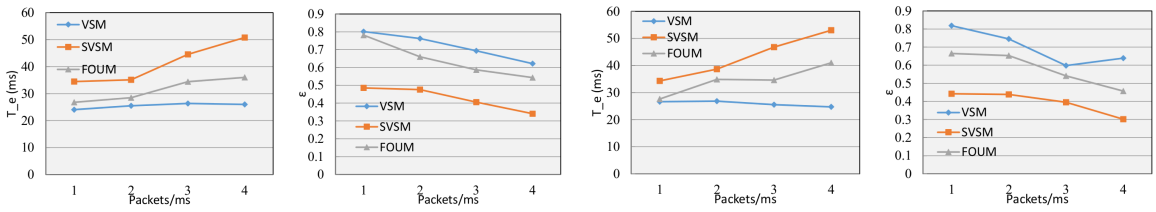
We first evaluate the three mechanisms for the first scenario. We vary the flow length among 4 and 5. The results are presented in Fig. 6. We can find that FOUM outperforms SVSM with respect to both two metrics. In particular, the throughput decrease of SVSM due to update is 20% more than that of FOUM. In addition, the value of T_e increases with the data rate, while the throughput decreases with the data rate in both SVSM and FOUM. This indicates that the update efficiency decreases as the network getting busier. We can also find that the update efficiency slightly decreases as the path increases.

The results for the second and third scenario are presented in Fig. 7 and Fig. 8, respectively. We obtains the similar observations that FOUM outperforms SVSM for both two metrics, and the differences get even bigger. These results well demonstrate the much better update efficiency of FOUM compared with SVSM.

VII. Related work

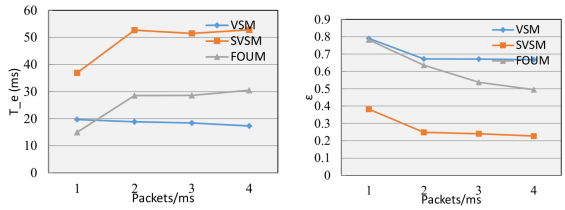
Reitblatt et al. [12, 13] are among the first to study the problem of update consistency in SDN. They propose a version-stamping update mechanism to ensure the consistent update. This scheme assigns a version number to each configuration, and stamps the packets entering the network at the ingress ports with their current version numbers(vn) in the headers. The update process is composed of four steps: (1) install the new policies on the internal switches while leaving the old ones in place, (2) once step 1 completes, install new policies on ingress switches and then stamp incoming packets with a new version number, (3) wait until all packets matching the old version number leave from the network, and (4) remove all the old policies.

McGeer al. [7] describes a new protocol for the update of OpenFlow networks. This protocol has the advantages of [12] as it saves the switch resources by keeping only one set of



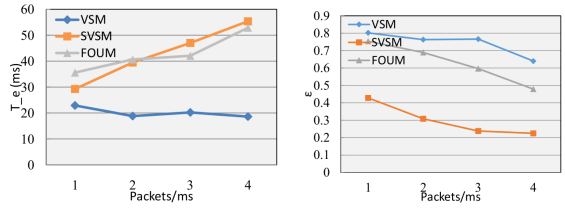
(a) T_e $(PathLength = 4)$ (b) ε $(PathLength = 4)$ (c) T_e $(PathLength = 5)$ (d) ε $(PathLength = 5)$

Fig. 6: Performance for single-path updates



(a) T_e (b) ε

Fig. 7: Performance for routing-handoff updates



(a) T_e (b) ε

Fig. 8: Performance for Load-balance updates

rules on each switch at any time. During update, the packets affected by the change of the rules will be sent to the controller and re-released into the network after the new rules are sent to the switch. However, the resource saving of the switch is at the cost of slower packet transmission and higher controller work load. Katta et al. [4] introduce new algorithms that trade the time of a consistent update for the rule-space overhead. They break the update into several rounds and each round completes part of the update. McGeer al. [8] proposes a new update protocol by deriving a logic circuit for the update sequence which requires neither rule-space overhead nor transferring the packets to the shelter during the update. Peresíni et al. [11] study a different consistent problem inside the controller but not switches.

The works mentioned above are elegant, and have the improvements of saving the space on the switch by trading the update time or the bandwidth. However, none of them take the security into consideration. Our proposal can guarantee the per-packet consistency under attacks from compromised links and even switches.

VIII. Conclusion

In this paper, we have presented a security and efficient update method for OpenFlow networks and verified the performance including update time efficiency and the bandwidth with a SDN testbed built on six servers. Our proposal can provider per-packet consistency under the risk of being attacked by compromised links. In addition, our proposal can help the controller locate the possible links or switches quickly when the the exception occurs during the update. The experiment results demonstrate the high efficiency of our proposal.

Bibliography

[1] A. Chakrabarti and G. Manimaran. Internet infrastructure security: A taxonomy. *Network, IEEE*, 16(6):13–21, 2002.

[2] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: scaling flow management for high-performance networks. In *Proc. of SIGCOMM*, pages 254–265, 2011.

[3] S. Hong, L. Xu, H. Wang, and G. Gu. Poisoning network visibility in software-defined networks: New attacks and countermeasures. NDSS, 2015.

[4] N. P. Katta, J. Rexford, and D. Walker. Incremental consistent updates. In *Proc. of 2nd HotSDN*, pages 49–54, 2013.

[5] D. Kreutz, F. Ramos, and P. Verissimo. Towards secure and dependable software-defined networks. In *Proc. of 2nd HotSDN*, pages 55–60. ACM, 2013.

[6] R. Mahajan and R. Wattenhofer. On consistent updates in software defined networks. Technical Report MSR-TR-2013-99, Microsoft Research, 2013.

[7] R. McGeer. A safe, efficient update protocol for openflow networks. In *Proc. of 1st HotSDN*, pages 61–66, 2012.

[8] R. McGeer. A correct, zero-overhead protocol for network updates. In *Proc. of HotSDN*, pages 161–162, 2013.

[9] 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.*, 38(2):69–74, 2008.

[10] P. Papadimitratos and Z. J. Haas. Securing the internet routing infrastructure. *Communications Magazine, IEEE*, 40(10):60–68, 2002.

[11] P. Peresíni, M. Kuzniar, N. Vasic, M. Canini, and D. Kostic. Of.cpp: consistent packet processing for openflow. In *Proc. of HotSDN*, pages 97–102, 2013.

[12] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proc. of SIGCOMM*, pages 323–334, 2012.

[13] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Consistent updates for software-defined networks: Change you can believe in! In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pages 7:1–7:6, New York, NY, USA, 2011. ACM.

[14] S. Waichal and B. Meshram. Router attacks-detection and defense mechanisms. *IJSTR*, 2(6), 2013.