This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Measuring the Consistency Between Data and Control Plane in SDN

Kai Lei, *Member, IEEE*, Guanjie Lin, Meimei Zhang, Keke Li, Qi Li, Xiaojun Jing, and Peng Wang

***Abstract*—Software Defined Networking (SDN) simplifies network control and management by decoupling the control plane from the data plane. However, the actual packet behaviors, conforming to the rules in the data plane flow tables, may violate the original policies in the controller due to the inconsistency between the data plane and control plane. To address this problem, we propose *2MVeri*, a framework for measuring *the consistency between the Data and Control plane*, defined as the consistency between the control plane policies and data plane rules. 2MVeri uses a modules, a Bloom filter and a two-dimensional vector as a tag which is inserted in the packet header and is updated in each switch that the packet traverses. By exploiting path information compressed in the tag, 2MVeri can verify the consistency between the data and control plane. Moreover, when verification fails, 2MVeri is able to localize the faulty switch. Experimental results show that in the k = 4 fat tree topology, the verification accuracy of 2MVeri is as high as 100%. In addition, when the actual path is inconsistent with the expected path, 2MVeri can locate the wrong switch with an accuracy of 99.8%.**

***Index Terms*—Software defined networking, consistency verification, fault localization.**

I. Introduction

AS AN emerging network paradigm, Software Defined Networking (SDN) decouples the control logic from forwarding devices in terms of that the controller dictates packet forwarding by installing rules in switches [1]–[3]. However, the *actual path* which is taken by packets may not be the same with the *expected path* that the controller wants them to go through due to the inconsistency between the control plane policies and data plane rules. Several factors, as listed below, can result in the inconsistency between the control plane and data plane.

- **Lack of acknowledgement.** In practice, there is no acknowledgement mechanism on a per-rule basis to ensure the success of rule installation process. Although the OpenFlow specification [4] does provide the Barrier command to indicate the completion of rule installations, this command is at batch level [5]. Moreover, even if the rules have not been installed properly, the switches can still send Barrier replies to the controller [6]–[8].
- **Switch software bugs and hardware failures.** According to the survey conducted among the subscribers to the NANOG[1] mailing list, the top two causes of network failures are switch software bugs and hardware failures [9]. For example, [6] shows that the HP ProCurve 5406zl switch lacks support for rule priority. The switch always treats the priorities of the rules installed later higher than those installed earlier, as a result, if the higher-priority rules are installed earlier, the packets may be wrongly forwarded according to the lower-priority ones.
- **Rule modification from external sources.** Besides the controller, the rules in the switches can be modified by other external sources as well. For example, a careless operator may misconfigure the switches and modify the original rules installed by the controller. Malicious attackers can utilize some 3rd-party tools (e.g. ONIE [10] and Fake LLDP Injection [11]) to take fully control over SDN switches and modify the rules. These modifications may not be noticed by the controller, thus causing the inconsistency between the control plane and data plane.
- **Network update.** In response to events such as network link failures, load changes and topology changes, the controller installs new rules in switches. However, due to the complicated network update process and the asynchronous communication of network update commands (e.g. FlowMod message in OpenFlow specification), the rules in the switches may not be updated on time [8], [12]–[14], resulting in the inconsistency between the control plane policies and data plane rules. This kind of inconsistency caused by network update is transient, which will be eliminated after the installation of new rules.

The inconsistency between the control plane and data plane can cause the network out of control. For example, the traffic engineering and access control policies could fail to function properly if the packets are forwarded to the undesired links or subnetworks. The uncertain network behaviors will

Manuscript received 9 May 2021; revised 19 February 2022 and 19 June 2022; accepted 8 July 2022; approved by IEEE/ACM Transactions on Networking Editor S. Ioannidis. This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant 62072012, in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2020B0101090003, in part by the Shenzhen Research Project under Grant JSGG20191129110603831, and in part by the Shenzhen Key Laboratory Project under Grant ZDSYS201802051831427. *(Corresponding author: Kai Lei.)*

Kai Lei, Meimei Zhang, and Keke Li are with the Shenzhen Key Laboratory for Information Centric Networking and Blockchain Technology (ICNLAB), School of Electronic and Computer Engineering (SECE), Peking University, Shenzhen 518055, China (e-mail: leik@pkusz.edu.cn).

Guanjie Lin is with the School of AI—Guangdong and Taiwan, Foshan University, Foshan 528225, China.

Qi Li is with the China Merchants Group, China.

Xiaojun Jing is with Surfilter Network Technology Company Ltd., Shenzhen 518057, China.

Peng Wang is with Now Karin Group Holding Ltd., China.

Digital Object Identifier 10.1109/TNET.2022.3193698

[1] North American Network Operators' Group.

1558-2566 © 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Foshan University. Downloaded on September 06,2022 at 13:38:10 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

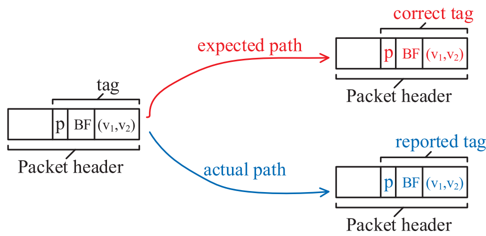

Fig. 1. Explanation of 2MVeri. BF is the Bloom filter. (v_1, v_2) is the two-dimensional vector and p is the modulus.

further result in performance degradation or even network fault (e.g. forwarding loops and packet dropping). It is therefore important to ensure the consistency between the control plane and data plane.

To solve the inconsistency problem, many algorithms and tools have been introduced in the past years. The proposed approaches can be classified into two groups: *in-band* solutions, which exploit tags and the information of control plane configurations to monitor the control-data plane consistency [15]–[17], and *out-of-band* solutions, which inspect packet forwarding behaviors using customized probe packets [5], [9], [18], [19]. However, both types of solutions have their own limitations. Out-of-band solutions can not be applied to frequently updated networks because of the slow generation speed of probe packets, while the existing in-band methods are either unable to localize the faulty switch [16], [17] or could introduce considerable overhead to store and represent the expected path information [15].Therefore, how to implement path detection accurately and efficiently under the premise of ensuring low overhead is a key issue faced by current SDN path detection related research, which is the research *motivation* of this paper.

In this paper, we propose 2MVeri, a new in-band framework to measure the control-data plane consistency [20]. As shown in Fig. 1, 2MVeri introduces a tag, which consists of a Bloom filter, a two-dimensional vector and a modulus, in the packet header. In data plane, each switch maintains a $2 * 2$ matrix which is assigned by the controller. Whenever a packet passes through a switch, its tag in the header is updated: the modulus is multiplied by the determinant of the switch matrix, and the vector is multiplied by the switch matrix. In order to avoid overflow which is caused by matrix multiplication, 2MVeri applies modular mechanism on the result of matrix multiplication and uses this result as the updated value of the vector. At the same time, the switch id is put into the Bloom filter. When the packet leaves the network, the values of the Bloom filter and the two-dimensional vector in the tag, as well as part of packet header information, are reported to the controller for verification. Afterwards, the tag is removed and the packet is delivered to the destination. In the control plane, the controller also computes the correct tag in the same way as the reported tag is updated, but with the matrices and ids of the switches in the expected path. By comparing the reported tag and the correct tag which is found by matching on the reported header information, 2MVeri is able to measure the control-data plane consistency.

Compared with the existing in-band solutions [15]–[17], 2MVeri only requires the controller to calculate and store the correct tags besides part of packet header information. In this way, 2MVeri incurs much smaller overhead both in storage and computation in the control plane. Moreover, 2MVeri does not need to modify the packet format, nor does it require any special hardware support, which is demanded by [16] and [17]. Last but not the least, since 2MVeri does not utilize probe packets to inspect packet forwarding behaviors, it can be applied to frequently updated networks while out-of-band solutions can not. In summary, the contributions of this paper can be summarized as follows:

- We have proposed 2MVeri, the first framework (to the best of our knowledge) that exploits matrix multiplication to measure the control-data plane consistency. Compared with other existing solutions, 2MVeri can be applied to more general cases and introduces much smaller overhead.
- We formulate a set of formal constraints that switch matrix, the modulus and the two-dimensional vector should satisfy and provide detailed and rigorous mathematical proof, which is the key to ensuring the correctness of 2MVeri theoretically.
- We have conducted comprehensive experiments to evaluate the performance of 2MVeri. Evaluations show that 2MVeri can achieve as high as 100% accuracy in consistency verification and 99.8% accuracy in fault localization, for fat tree topology k=4.

In the rest of this paper, we will first discuss some related work in detail in Section II. Then, we will give an overview of 2MVeri in Section III and present the design details in Section IV. Later, the design rationale of the framework is shown in Section V. Our experiments come after in Section VI. After the evaluation, in Section VII, we conclude the paper.

II. Related Work

A. In-Band Works

In-band solutions directly embed verification information in the data traffic and do not utilize any dedicated traffic (i.e. probe traffic) to detect the control-data plane consistency.

VeriDP [15] inserts a 16-bit Bloom filter as the tag into packet headers, and during the packet forwarding process, the tag is updated. When a packet is about to leave the network, the tag and the packet header are reported to the controller. In the control plane, a server alongside the SDN controller is responsible for constructing the *path table* which is used to record the expected paths hop-by-hop for every flow from each input and output port pair. Each path table entry consists of: (1) a pair of input and output port pair $(inport, outport)$; (2) a set of packet headers representing packets which enter the network at the input port $inport$ and leave the network at output port $outport$; (3) a set of correct tags which are calculated if the corresponding packets take the expected paths.

By comparing the reported tag and the corresponding path table entry which is found by matching the reported packet

Authorized licensed use limited to: Foshan University. Downloaded on September 06,2022 at 13:38:10 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

header, VeriDP can verify whether the packet forwarding behavior is consistent with the policies in the controller. Since VeriDP uses the *path table* whose storage and construction will cause large overhead to record expected path information, the storage and computational overhead of VeriDP is relatively larger, obviously.

The work in [16] focuses on the rule verification problem under the possible rule modification attack, in which the adversary can modify flow table rules by utilizing implementation vulnerabilities of switch OSes and control channels. To cope with this attack, the author proposes a new security primitive named rule enforcement verification (REV), which exploits message authentication code (MAC) to check whether rules have been installed successfully by switches or not. To avoid the heavy switch-to-controller traffic incurred by standard MAC, REV uses a new compressive MAC, which allows switches to compress MACs before reporting to the controller.

In general, REV adopts a similar idea as VeriDP does to detect the control-data plane consistency: in entry switches, a tag is attached to the packet when it enters the network. Switches that are located in the forwarding path of the packet will update the tag and when the packet leaves the network, the exit switch will report the tag to the controller for verification. However, the composition and the generation of the tag in REV is much more complicated than in VeriDP. The tag in REV is generated by applying a series of cryptographic hash function with different keys, which are only known to the controller and the corresponding switches. Besides, the tag is put into another packet header named REV header, which sits in-between the IP header and the TCP/UDP header.

Obviously, the cryptographic operations through which REV tag is calculated may be computationally intensive and need the support of the underlying hardware. Moreover, this method needs to change the packet format, which may hinder its deployment. Last but not the least, if verification fails, REV does not provide network operators with any method to figure out the compromised or faulty switch.

Cisco proposes two cumulative schemes to verify transit of network traffic by exploiting in-band metadata of the packet [17]. The first one is encryption scheme, in which each switch keeps an unique cryptographic key that is distributed by the controller and the egress switch is configured with information for all these cryptographic keys. The ingress switch inserts a *SEQ* field, which contains a timestamp when the switch receives a packet, and a *VERIFY* field, which is initialized to 0, in each packet. The value in *VERIFY* filed is updated by each switch that the packet traverses based on the result of *XOR* operation of the initial value in *VERIFY* field and the encryption result of the value in *SEQ* field with the unique key. The egress switch only needs to repeat the process with the keys of switches in the expected path to get a correct *VERIFY* field value. By comparing the correct value with the existing one in the packet, the egress switch could determine whether the packet has taken an expected path or not.

Like [16], the cryptographic operations may need the support of the underlying hardware and it does not consider faulty switch localization. Moreover, since *XOR* operation is not order independent, if the packet passes the same set of switches as the expected path but with different order, this scheme will not be able to detect the inconsistency of the control plane and the data plane.

The second scheme makes use of Shamir's Secret Sharing Scheme (SSSS) [21] to verify transit of network traffic. In this scheme, each network switch in the expected path maintains a piece of the cryptographic secret and the egress switch has access to the complete secret. Each time a switch receives a packet, it will update the verification information in the packet with its piece of the cryptographic secret. When the packet arrives at the egress switch, the switch will try to recover the complete secret according to the many pieces of cryptographic secret which is updated by each switch. By comparing the recovered secret with the authentic secret which is distributed by the controller, the egress switch is able to check whether the data plane is consistent with the control plane or not. The main drawback of this scheme is that it does not take faulty switch localization into account likewise.

B. Out-of-Band Works

Out-of-band solutions inspect the forwarding behaviors of the dedicated probe traffic which is injected into the network to verify the control-data plane consistency.

ATPG [9] reads the device configuration files and FIBs (Forward Information dataBase) to generate minimum number of probe packets with the intention of exercising every rule in the data plane and testing every link in the network. More importantly, it takes a substantial time for ATPG to generate probe packets.

Although Monocle [18] addresses the rule priority problem faced by ATPG [9], the generation speed of probe packets still takes a massive time (e.g. 43 seconds for 10K real rules [15]). Besides, by acting as a proxy between the controller and switches, Monocle intercepts all rule modifications issued to switches, and sends update acknowledgements to the controller, which exacerbates the latency between the controller and switches further.

RuleScope [5] injects customized probe packets to detect rule missing faults and priority faults, besides, RuleScope tries to uncover actual data-plane forwarding states after detecting faults. Although RuleScope employs various techniques to enhance detection effiency, it has a slow generation speed of probe packets as well (e.g. more than 300 seconds for 320 synthetic rules [15]).

WedgeTail [22], an Intrusion Prevention System designed to secure the SDN data plane. WedgeTail's core detection and response techniques are platform independent and network dynamics do not alter these. To retrieve the expected trajectories, WedgeTail intercepts the relevant OpenFlow messages exchanged between the control and data plane and maintains a virtual replica of the network. This virtual replica is processed by its integrated Header Space Analysis (HSA) component to calculate the expected packet trajectories. Strictly speaking, this paper is not the same as WedgeTail, which is a heavy-weight intrusion detection and prevention system, while this paper focuses on the path detection problem in SDN.

PAZZ [23] adds network state fuzzing based on out-of-band work to its consistency checks, and PAZZ periodically fuzzes

Authorized licensed use limited to: Foshan University. Downloaded on September 06,2022 at 13:38:10 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

the header space to help PAZZ verify consistency. PAZZ combines production flow with active probes, and even if there may be a comprehensive effect, the computational overhead cannot be ignored, and the accuracy remains to be verified.

P4CONSIST [24] generates active probe-based traffic continuously or periodically as an input to the P4 SDNs to check whether the actual behavior on the data plane corresponds to the expected control plane behavior. In P4CONSIST, the control plane and the data plane generate independent reports which are later, compared to verify the control-data plane consistency. Experiments with P4CONSIST are time-consuming and show that P4CONSIST can verify the control-data plane consistency with 60k rules per switch within a minimum time of 4 minutes.

III. Design Overview

A. Principles and Challenges

1) Checking Flow Tables or Packet Forwarding Behaviors: The easiest way to check the control-date plane consistency is to dump all rules from switches and compare them with the policies, however, this method is not suitable for the network with complex status and frequent updates, what's more, one of the reasons for the inconsistency between the data and the control layer is the network update. From the other point of view, rules in data plane flow tables are directly reflected on the forwarding behaviors of packets, which means that *we can detect the control-data plane consistency by checking packet forwarding behaviors instead of dumping all rules from switches.*

To inspect packet forwarding behaviors, we first need to determine whether to construct probe packets and verify their forwarding behaviors or to sample and tag the real traffic for verification.

2) Probes or Real Traffic: If we use probe packets and inspect their forwarding behaviors, the probe packets need to be carefully designed so that they only trigger the specific rules while not being affected by other rules. As a result, constructing the appropriate probe packets becomes a complicated and time-consuming task [5], [9], [18], [19]. Moreover, even with careful design, the forwarding behaviors of probe packets can be still different from the real packets in some strict senses [15]. Given these limitations, *we decide to sample and tag the real traffic instead of using probe packets to measure the control-data plane consistency.*

After making this decision, we need to consider how to minimize the overhead as much as possible, as well as achieve high accuracy and efficiency in detection at the same time. Then, we face with the following challenges:

Challenge 1. Efficient and informative tag design. The tags are used to compress the actual path information and are updated when packets traverse the network. We need an efficient and informative tag design so that: 1) the overhead of tag storage, transmission and computation is tolerable; 2) the actual path information compressed in the tag can be used for verification and localization; 3) the inserted tags do not change the format of the packet.

Challenge 2. Accurate and efficient representation of the expected paths. Besides the actual path information compressed in the tag, the information of the expected paths is also needed for verification. However, the cost to store the expected paths hop-by-hop for every flow (e.g. the path table in [15]) is relatively large. Thus, we need an accurate and efficient way to record the expected paths.

To address the above challenges, *we use a 16-bit Bloom filter and a two-dimensional vector as the tag. In this case, the information of the expected paths can be directly represented by the tag as well.* Specifically, in 2MVeri, each switch maintains a $2 * 2$ characteristic matrix which is assigned by the controller. During the packet forwarding process, the vector in the tag is multiplied by the switch matrix and the switch id is put into the Bloom filter. Due to the irreversibility of matrix multiplication, the path information can be exclusively represented by the corresponding tag.

However, this kind of way to represent the path information is highly possible to cause overflow since matrix multiplication is required at each switch. Once overflow happens, the verification and localization accuracy will decline dramatically. To avoid this undesired situation, 2MVeri applies modular mechanism at each switch and now *the tag consists of a 16-bit Bloom filter, two-dimensional vector and a modulus.*

During the packet processing procedure in each switch, instead of directly updating the vector to the result after the matrix multiplication, 2MVeri applies modular mechanism on the matrix multiplication result and puts this value into the header. Despite the avoidance of overflow, modular mechanism loses part of path information in nature, since neither switches keep quotients nor the controller knows them, and increases the difficulty of recovering actual paths that packets take. The problem of finding appropriate moduli which do not affect the accuracy of recovering actual paths becomes urgent. Thus, here comes the third challenge:

Challenge 3. Appropriate moduli design. The overhead of maintaining all quotients of each multiplication is unbearable, therefore, we need appropriate moduli so that there is no need to store quotients to recover the actual path. *2MVeri associates the value of moduli to the determinants of corresponding matrices, in which case the accuracy of verification and localization is guaranteed in the absence of quotients.* We will discuss the modulus design in detail in Section V.

B. 2MVeri Overview

As shown in Fig. 2, 2MVeri consists of three subsystems: packet processing subsystem, verification subsystem, and localization subsystem.

1) Packet Processing Subsystem: This subsystem is responsible for sampling, tagging and reporting packets to the controller. Here we classify switches into three categories: entry switches, internal switches, and exit switches. Entry switches and exit switches are edge switches where packets enter and leave the network, respectively. Internal switches, on the other hand, interconnect entry switches and exit switches. Instead of tagging and verifying every packet in the network, entry switches sample packets based on flows. Once a packet is received and sampled by the entry switch, an additional tag will be initialized and inserted into its header. Whenever the sampled packet passes through switches along the forwarding

Authorized licensed use limited to: Foshan University. Downloaded on September 06,2022 at 13:38:10 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

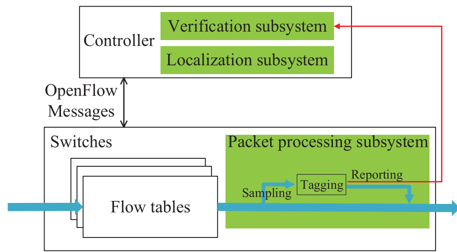

Fig. 2. System architecture. The green parts belong to 2MVeri.

path, the tag will be updated with a new value calculated based on the characteristic matrices and ids of the switches (see Section IV). Finally, before the sampled packet leaves the network, the values of the Bloom filter and the two-dimensional vector in the tag as well as part of packet header are reported to the control plane for further processing. Afterwards, the tag is removed and the packet is delivered to the destination.

2) Verification Subsystem: The controller computes correct values of the Bloom filter and the two-dimensional vector with the information of switches in expected paths. After receiving the values of the Bloom filter and the vector that are reported by the exit switch, the controller compares the reported values with the correct values which are found by matching on the reported packet header. If and only if the reported values and the correct values of the Bloom filter and the vector are identical, respectively, 2MVeri can affirm that the actual path is consistent with the expected one.

3) Localization Subsystem: When the verification fails, the localization subsystem will recover the actual forwarding path of the packet based on the information compressed in the tag to localize the wrong switch.

IV. Design Details

A. Packet Processing Subsystem Design

2MVeri employs an additional tag in the packet header to compress path information. Specifically, a two-dimensional vector, which consists of two 32-bit random numbers, a 16-bit Bloom filter and a 32-bit modulus, are introduced as the tag. The two-dimensional vector, the 32-bit modulus and the 16-bit Bloom filter are inserted in the three 32-bit mpls_label fields and the 16-bit VLAN field, respectively. In data plane, each switch maintains a $2*2$ characteristic matrix, which is assigned by the controller when the switch is initialized.

An example for packet processing procedures is illustrated in Fig. 3. 2MVeri associates a *sampling interval* T_s^f with each flow f that is identified by the 5-tuple of source and destination IP addresses, ports, and transport protocol. Entry switches use a single bit in the IP DSCP field to indicate whether the packet is sampled or not and keep a hash table to store the last time T_{last}^f when each active flow f is sampled. When entry switches receive a packet of flow f at time t, if $t - T_{last}^f \geqslant T_s^f$, the packet is sampled and T_{last}^f, whose initial value is 0, is updated to t. Only the sampled packet is inserted with a tag, which is initialized in the entry switch and updated in each switch that the packet traverses based on switch matrix and switch id. Sampled packets of the same flow share the identical initial value of the vector, which is reported to the controller by the entry switch after sampling a packet from a new flow. The controller and entry switches both use a hash table to store the initial value of the vectors for all active flows. Moreover, the controller also needs to record the id of the entry switch which reports the corresponding initial value of flows, and entry switches are responsible for sending the value in the TTL field in the packet header to the controller as well when receiving a packet from a new flow to assist localization (see Section IV-C).

In Fig. 3, M_1, M_2, M_3 are the characteristic matrices of switch S_1, S_2, S_3, respectively. The brown rectangles are packets, where "hdr" represents packet header and the shadowed part is the payload.[2] The entry switch S_1 initializes the tag by setting the value of the 16-bit Bloom filter BF and the 32-bit modulus p to 0 and 1, respectively, and assigning (v_1, v_2) as the initial value of the vector to the sampled packet based on flow. If the sampled packet comes from a new flow, the initial value of the vector and the value in TTL field are sent to the controller. Each switch $S_i(i = 1, 2, 3)$ that receives the sampled packet updates the tag with its switch id ID_i, and matrix M_i: $p = p * det(M_i), BF = BF \cup BF(ID_i), (v_{2i+1}, v_{2i+2}) = (v_{2i-1}, v_{2i}) * M_i \% p$. Before the packet leaves the network, the exit switch S_3 removes the tag and reports the values of Bloom filter and the two-dimensional vector along with the value in TTL field to the controller for verification and localization.

The packet processing algorithm is shown in Algorithm 1, entry switches are responsible for initializing the tag (Line 2-4), as well as sending the initial value of the vector and the value in the TTL field for each flow to the controller (Line 5-7). Each switch that the packet goes through needs to update the tag (Line 9-10). Before the packet is dropped or delivered to the destination, the Bloom filter, the two-dimensional vector and the value in TTL field are reported to the controller. Finally, the tag is removed from the packet (Line 12-13).

B. Verification Subsystem Design

In the controller, the correct values of Bloom filter and vectors can be calculated and stored beforehand based on the matrices of the switches along the expected path. For verification, the controller only needs to check whether the reported Bloom filter and vector are the same with the correct Bloom filter and vector, respectively. Verification succeeds if and only if they are equal. The verification algorithm is presented in Algorithm 2.

For a concrete example, consider the network in Fig. 4, in which we suppose that the expected path of a packet is $S_1 \rightarrow S_2 \rightarrow S_4$ while the actual path is $S_1 \rightarrow S_3 \rightarrow S_4$. Based on the expected path, the correct value of Bloom filter can be computed as: $BF_c = BF(S_1) \cup BF(S_2) \cup BF(S_4)$. The correct value of the two-dimensional vector can be calculated

[2]We enlarge the packet header just to make the packet processing procedure more clearly, not meaning the packet header takes larger space than the payload physically.

Authorized licensed use limited to: Foshan University. Downloaded on September 06,2022 at 13:38:10 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

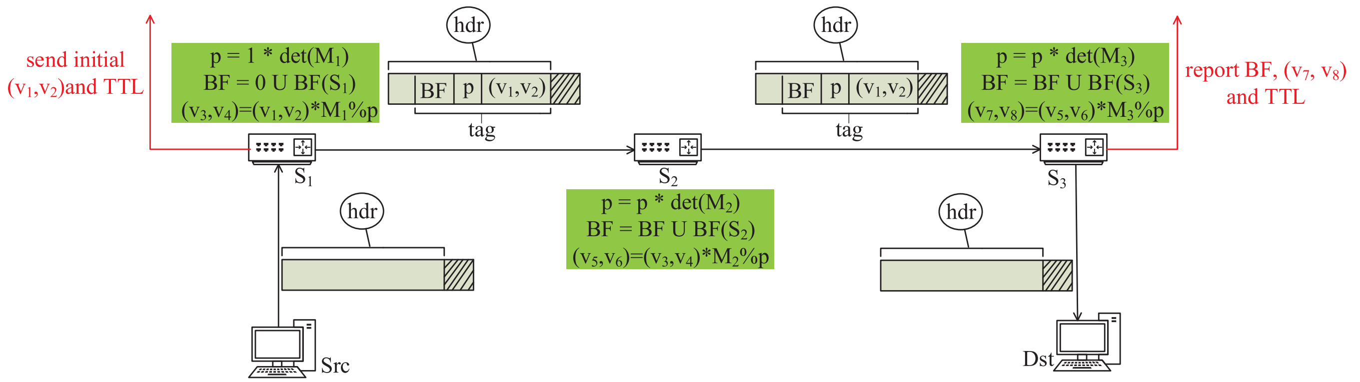

Fig. 3. Example for packet processing. S_1, S_2, S_3 are the entry switch, internal switch and exit switch, respectively. Green components are the packet processing subsystems. $(v_{2i+1}, v_{2i+2})(i = 1, 2, 3)$ is the updated vector, whose initial value is (v_1, v_2), in each switch S_i. BF is the Bloom filter and p is the modulus.

Algorithm 1 Packet Processing Algorithm

Input:

ID_i: The switch id of $S_i (i \in \mathbb{Z}^+)$.
M_i: The $2 * 2$ characteristic matrix of the switch S_i.
(v_1, v_2): Initial value of the two-dimensional vector.
(v_{2i+1}, v_{2i+2}): Updated value of the vector in switch S_i.
TTL_{init}: TTL value when the packet enters the network.
TTL_{last}: TTL value when the packet leaves the network.
BF: The value of the Bloom filter.
p: The modulus in the packet header.
tag: The tag in the packet header.

1: **for** each switch S_i **do**
2: **if** *The input port is an edge port* **then**
3: $BF = 0$; $p = 1$;
4: assign (v_1, v_2) to the packet based on flow;
5: **if** the sampled packet comes from a new flow **then**
6: send (v_1, v_2) and TTL_{init} to the controller;
7: **end if**
8: **end if**
9: $p = p * det(M_i)$; $BF = BF \cup BF(ID_i)$;
10: $(v_{2i+1}, v_{2i+2}) = (v_{2i-1}, v_{2i}) * M_i \ \% \ p$;
11: **if** *The output port is an edge port OR drop the packet* **then**
12: Report BF, (v_{2i+1}, v_{2i+2}) and TTL_{last} to the controller;
13: Remove tag from the packet;
14: **end if**
15: **end for**

as: $(V_c^1, V_c^2) = (v_1, v_2) * M_1 \% det(M_1) * M_2 \% (det(M_1) * det(M_2)) * M_4 \% (det(M_1) * det(M_2) * det(M_4))$. However, the reported values of Bloom filter and vector are: $BF_p = BF(S_1) \cup BF(S_3) \cup BF(S_4), (V_p^1, V_p^2) = (v_1, v_2) * M_1 \% det(M_1) * M_3 \% (det(M_1) * det(M_3)) * M_4 \% (det(M_1) * det(M_3) * det(M_4))$.

If there is no modulo operation, then, the reported tag is likely to be different from the correct tag, causing the failure of the verification. The reason is straightforward. If the packet takes a different path instead of the expected path, the reported tag will be different from the correct one with large probability. Even if the packet passes the same set of switches as the expected path but with different order, verification will still fail due to the irreversibility of matrix multiplication.

Algorithm 2 Verification Algorithm

Input:

BF_c: The correct value of the Bloom filter.
BF_p: The reported value of the Bloom filter.
(V_c^1, V_c^2): The correct value of the vector.
(V_p^1, V_p^2): The reported value of the vector.

Output:

return *consistent* or *inconsistent*.

1: **if** $BF_c = BF_p$ && $(V_c^1, V_c^2) = (V_p^1, V_p^2)$ **then**
2: return *consistent*;
3: **else**
4: return *inconsistent*;
5: **end if**

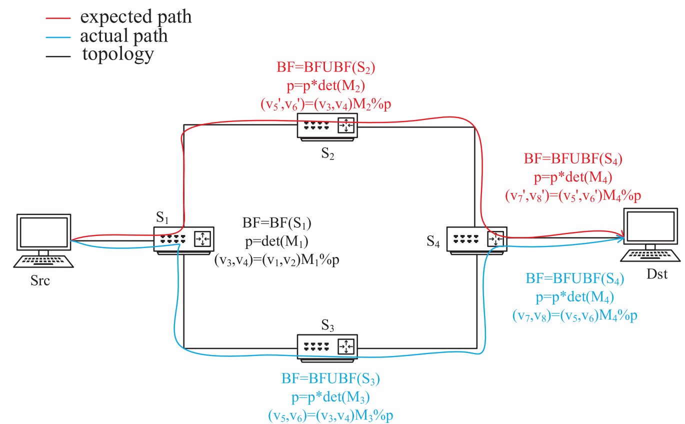

Fig. 4. Example for verification and localization. (v_{2i+1}, v_{2i+2}) and $(v'_{2i+1}, v'_{2i+2})(i = 1, 2, 3)$ are the updated vectors, whose initial values are (v_1, v_2), in each switch. BF is the Bloom filter and p is the modulus. M_1, M_2, M_3, M_4 are the matrices of switch S_1, S_2, S_3, S_4, respectively.

Nevertheless, in order to avoid overflow that is caused by the matrix multiplication, we have to apply modular operation, which maps the result after multiplication to the interval from

Authorized licensed use limited to: Foshan University. Downloaded on September 06,2022 at 13:38:10 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

0 to $p-1$, where p is the modulus. In order to guarantee the reported tag to be as different as possible from the correct tag when inconsistency happens and considering the false positive of the Bloom filter, it would be better to ensure that the reported vector is different from the correct vector, which means that the modulus p can not be too small. Since if p is too small and the multiplication result is large enough, then it is of great possibility that the mapping result of the actual path is identical with that of the expected path, causing the reported vector being the same with the correct vector, thus increasing the false negative rate of verification. In 2MVeri, we adjust and enlarge the value of modulus p based on cumulative multiplication of matrix determinant. Evaluations in Section VI show that this kind of method could achieve almost the same verification and localization accuracy as no modular mechanism is applied when there is no overflow.

C. Localization Subsystem Design

Before we move to the detailed design of the fault localization subsystem, we use an example to demonstrate the idea of the algorithm firstly.

Looking back to Fig. 4 where the packet takes a different path from the expected path, causing the failure of the verification. However, we can not simply blame S_3 for the deviation from the expected path, because the upstream switch S_1 wrongly forwards the packet. To localize the faulty switch, we need to recover the actual forwarding path. After recovering the actual path, localization becomes easy: the faulty switch is the first switch that goes wrong.

Firstly, to make things simpler, consider the case where there is no modular mechanism. Based on the reported vector (v_7, v_8), the controller could calculate $P = (v_7, v_8) * (M_4)^{-1}$, in which P equals to $(v_1, v_2) * M_1 * M_3$ and consists of two positive integers. Then, according to the topology, the adjacent switches of S_4 are S_2 and S_3. Since switches in the actual path record their switch ids in the reported Bloom filter, if we can find the id of the possible switch in the reported Bloom filter, the switch is likely in the actual path. So we can check whether S_2 and S_3 are in the reported Bloom filter or not. On one hand, S_3 could be found in the reported Bloom filter out of question, on the other hand, however, due to the *false positive* of the Bloom filter, S_2 may be found in the reported Bloom filter as well. Therefore, to eliminate the false positive of the Bloom filter, we need to calculate $P * (M_i)^{-1}$, where M_i is the matrix whose switch is found in the reported Bloom filter, which are S_2 and S_3 in our case. Only the product consists of two positive integers, can we confirm that the switch is in the actual path. Apparently, $P_1 = P*(M_3)^{-1}$ must be two positive integers, since $P_1 = P*(M_3)^{-1} = (v_1, v_2)*M_1$. On the other hand, $P * (M_2)^{-1}$ will be fractions with large probability. We can recover the actual forwarding path by continuing this process until the product equals to the initial vector, that is, (v_1, v_2). In this example, the process is terminated after calculating $P_1 * (M_1)^{-1}$, since $P_1 * (M_1)^{-1} = (v_1, v_2)$.

The process of recovering the actual path above sounds straightforward up till now, however, the modular mechanism makes things more complicated:

1. When there is no modular mechanism, we carry out the multiplications of vectors and the inverse of matrices of possible switches to eliminate false positive of Bloom filter, and if the result of the multiplication consists of integers, the switch is located in the actual path; otherwise, it is not. Nevertheless, if we apply modular operations, the result of the multiplication may not consist of integers even if the switch is located in the actual path. Thus, we can not use integer as a standard to determine whether the switch is located in the actual forwarding path or not any more, then, how can we decide whether the switch is in the actual path or not? Or, can we find a set of moduli which could make the result of the multiplication be integers if the switch is located in the actual path, just as when there is no modular mechanism is applied?

2. Assume we could find an effective method to eliminate the false positive of the Bloom filter, when should we stop recovering the actual path? If there is no modular mechanism, the process of recovering suspends when the value of the vector is the same as the initial one. However, with modular operations, the value of the vector may not equal to the initial one even though we have recovered the actual path.

To solve the first question, we design a sophisticated solution that is used to determine the values of moduli (see Section V), to make sure that the multiplication results of vectors and the inverse of matrices consist of integers if switches are in the actual path. That is to say, the localization technique when no modular mechanism is applied could be reused to a great extent.

On the other hand, since the value in the TTL field in ip header will decrease by 1 every time the packet passes a switch, so we could use the difference of the TTL field between the one when the packet enters the network and the one when it leaves the network to denote the number of switches that the packet passes. Once the number of switches which are located in the actual path is known, we are able to restore the actual forwarding path by continuing matrix multiplication process until the number of recovered hops equals to the number of switches in the actual path. Moreover, to improve the accuracy of localization, we need to check whether the entry switch which is located in the recovered actual path is the same with the entry switch that reports the initial value of the vector of the corresponding flow (see Section III). If and only if the two switches are the same, can we confirm that we have recovered a correct actual path.

Shown in Algorithm 3, we first determine whether we have recovered an actual path or not by checking the length of the *actualPath* which stores the recovered switches and is initialized to be an empty stack. If the number of recovered switches equals to the number of switches that the packet passes, an actual path has been recovered (Line 1- Line 8). Otherwise, we continue to carry out the recover process (Line 9-17). If we have restored an actual path, we need to check the correctness of the recovered actual path by verifying whether the entry switch that we recovered is the same with the one that reports the initial value of vector or not. If they are the same, then the algorithm regards this actual path as the correct one and returns this actual path, in addition to setting

Authorized licensed use limited to: Foshan University. Downloaded on September 06,2022 at 13:38:10 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Algorithm 3 Localization Algorithm

Input:

A: The adjacency matrix of the network topology.
$ID_{current}$: The id of the current switch.
ID_{entry}: The id of the entry switch that reports initial value to the controller.
BF: The value of the reported Bloom filter.
V: The value of the reported vector.
$TTL_{difference}$: The difference of TTLs.
M_i: The $2*2$ characteristic matrix of switch S_i.
$actualPath$: The actual path that the packet takes.
$found$: Indicate whether an actual path has been recovered or not. Initialized to false.

Output:

$actualPath$: The actual path that the packet takes.

1: **if** $actualPath$.size()=$TTL_{difference}$ **then**
2: **if** $actualPath$.top()=ID_{entry} **then**
3: $found$=true
4: return $actualPath$
5: **else**
6: return
7: **end if**
8: **end if**
9: **if** $BF(ID_{current}) \cap BF = BF(ID_{current})$ and $V * M^{-1}_{ID_{current}}$ consists of integers **then**
10: $actualPath$.push($ID_{current}$)
11: **for** Switch i satisfying $A(ID_{current}, i) = 1$ **do**
12: Localization(A, i, ID_{entry}, BF, $V * M^{-1}_{ID_{current}}$, $TTL_{difference}$, M, $actualPath$)
13: **end for**
14: **if** $found$=false **then**
15: $actualPath$.pop()
16: **end if**
17: **end if**

found to true (Line 2-4); otherwise, we need to go back to seek other switches (Line 5-7).

Recovering process starts by checking whether the switch both exists in the reported Bloom filter and the product of the vector and the inverse of its matrix consists of two positive integers. The switch is in the actual path only when the two conditions are met and we push this switch into the *actualPath* (Line 9-10). Then, we continue to find more possible switches which are adjacent to the current switch (Line 11-13). If we can not find a valid actual path which contains the current switch, we need to pop the current switch from the *actualPath* and roll back to the previous hop to find other possible switch (Line 14-16).

V. Design Rationale

To ensure the correctness and accuracy of 2MVeri, the matrices of the switches, the moduli and the vector should be carefully designed. In this section, we discuss the rationale behind the framework.

A. Moduli Design

As stated before, in order to avoid overflow, 2MVeri introduces modular operations after matrix multiplication. However, the modular mechanism brings about a new question which is critical to the correctness and performance of verification and localization algorithms: how to recover actual paths and localize faulty switches accurately with incomplete path information caused by the loss of quotients in each modular operation.

In order to overcome this problem, we design an intricate solution which is used to determine the values of moduli. The scheme could ensure that the multiplication result of the post-modulo vector and the inverse of the corresponding matrix consists of integers so that there is no need to restore the pre-modulo vector when recovering the actual forwarding path of the packet.

For example, refer to the left subfigure in Fig. 5 where S_f is the last-hop switch both in the actual path and the expected path. S_m and S_n are the previous hops of S_f in the actual path and the expected path, respectively. In this figure, the value of the vector in the packet header is (v_1, v_2) when the packet arrives at S_m and are updated to (v_5, v_6) after passing S_m and S_f, following the procedure mentioned in Section IV. Since the reported vector and Bloom filter are not the same as the correct ones, the controller needs to carry out localization algorithm to find out the faulty switch. Localization process is shown in the right subfigure and our scheme that is used to determine values of moduli should satisfy:

1. $(v_5, v_6)*M_f^{-1}$ and $(v_3', v_4')*M_m^{-1}$ both consist of integers. That is to say, the modular mechanism should ensure that the multiplication result of the post-modulo vector and the inverse of the corresponding matrix consists of integers, which means we could simply recover the actual path without enduring the possible error and extra computational overhead that are brought by the quotients restoring process.

2. $(v_3', v_4') * M_n^{-1}$ does not contain two integers as much as possible at the same time. It is obvious that the reported vector could be the same with the correct vector even if inconsistency happens, since modular operation maps the multiplication result to the interval from 0 to $p-1$. The larger value p is, the more accurate verification and localization will be. 2MVeri adjusts and enlarges the value of the modulus p based on cumulative multiplication of matrix determinant. Evaluations in Section VI show that this kind of method could achieve almost the same verification and localization accuracy as no modular mechanism is applied when there is no overflow.

Here we concentrate on the first property that moduli are supposed to satisfy. Using the notations shown in Table I, the problem of finding the best set of moduli could be described as follows:

Problem 1: Given

$$(v_{2i+1}, v_{2i+2}) = (v_{2i-1}, v_{2i}) * M_i \% p_i (i = 1, 2, 3 \ldots n).$$

Find $p_i (i = 1, 2, 3 \ldots n)$ such that

$$(v_{2i+1}, v_{2i+2}) * M_i^{-1}$$

consists of integers.

To find such $p_i (i = 1, 2, 3 \ldots n)$, we first introduce a lemma shown in Lemma 1.

Authorized licensed use limited to: Foshan University. Downloaded on September 06,2022 at 13:38:10 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

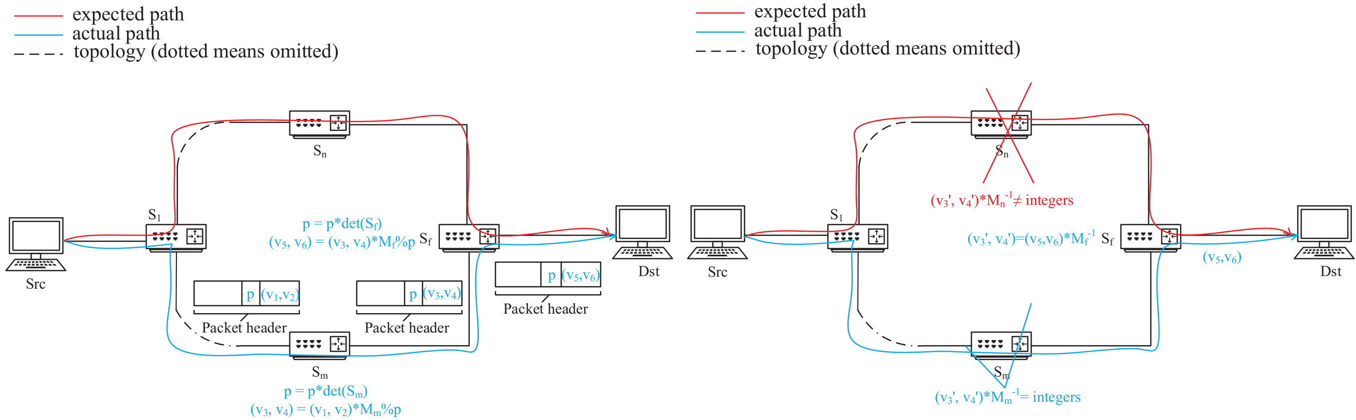

Fig. 5. Example to illustrate moduli design. The left subfigure is the packet processing procedure and the right subfigure shows the process of localization. The Bloom filter is not shown in the figure. p is the modulus and M_m, M_n, M_f are the characteristic matrices of switch S_m, S_n and S_f, respectively.

TABLE I

NOTATIONS AND CORRESPONDING MEANINGS

Notation	Meaning
n	The number of switches in the actual path.
$S_i(i = 1, 2, 3...n)$	The ith switch in the actual path.
$M_i(i = 1, 2, 3...n)$	The matrix of S_i.
$p_i(i = 1, 2, 3...n)$	The updated modulus in switch S_i.
(v_1, v_2)	The initial value of the vector.
$(v_{2i+1}, v_{2i+2})(i = 1, 2, 3...n)$	The value of the vector after passing S_i.

Lemma 1: If

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, p = det(A), k \in \mathbb{Z},$$
$$D_1 = (u_1, v_1), D_2 = D_1 * A\%(kp), D_3 = D_2 * A^{-1}.$$

Then there exists $M, N \in \mathbb{Z}$, such that:

$$D_3 = D_1 + k(M, N)$$

Proof:

$$\begin{aligned} D_2 &= D_1 * A\%(kp) \\ &= (u_1, v_1) * \begin{bmatrix} a & b \\ c & d \end{bmatrix} \%(kp) \\ &= ((au_1 + cv_1)\%(kp), (bu_1 + dv_1)\%(kp)) \end{aligned}$$

Let $D_3 = (u_3, v_3)$, then,

$$\begin{aligned} (u_3, v_3) &= D_2 * A^{-1} \\ &= D_2 * \frac{1}{p} * \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}. \\ u_3 &= \frac{1}{p} * (d((au_1 + cv_1)\%(kp)) \\ &\quad - c((bu_1 + dv_1)\%(kp))) \\ v_3 &= \frac{1}{p} * (-b((au_1 + cv_1)\%(kp)) + a((bu_1 \\ &\quad + dv_1)\%(kp))) \end{aligned}$$

Let $(au_1 + cv_1)/(kp) = m, (bu_1 + dv_1)/(kp) = n$, then

$$\begin{aligned} u_3 &= \frac{1}{p} * (d(au_1 + cv_1 - mkp) - c((bu_1 + dv_1 - nkp))) \\ &= \frac{1}{p} * ((ad - bc)u_1 + (cn - dm)kp) \\ &= \frac{1}{p} * (u_1 + (cn - dm)k) * p \\ &= u_1 + (cn - dm)k \\ &= u_1 + Mk \\ v_3 &= v_1 + (an - bm)k \\ &= v1 + Nk \\ (M &= cn - dm, N = an - bm.) \end{aligned}$$

Thus, $D_3 = D_1 + k(M, N).M, N \in \mathbb{Z}$ □

We prove the following proposition, which is shown below, based on the lemma above.

Proposition 1: If

$$\begin{aligned} A_1 &= \begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} A_2 = \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} \\ p_1 &= det(A_1), p_2 = det(A_1) * det(A_2), \\ C_1 &= (x_1, y_1), C_2 = C_1 * A_1\%p_1, C_3 = C_2 * A_2\%p_2, \\ C_4 &= C_3 * A_2^{-1}, C_5 = C_4 * A_1^{-1} \end{aligned}$$

then when C_1 consists of two integers, C_5 is also an integer vector.

Proof: To prove the proposition above, we only need to prove that there exists $M, N \in \mathbb{Z}$, and:

$$C_5 = C_1 + (M, N)$$

According to the Lemma 1, since

$$\begin{aligned} C_2 &= (x_2, y_2) \\ C_3 &= C_2 * A_2\%(det(A_1) * det(A_2)) \\ C_4 &= C_3 * A_2^{-1} \end{aligned}$$

then,

$$\begin{aligned} C_4 &= C_2 + det(A_1) * (M_1, N_1) \\ &= C_2 + p_1 * (M_1, N_1), M_1, N_1 \in \mathbb{Z} \\ C_5 &= C_4 * A_1^{-1} \\ &= (C_2 + p_1 * (M_1, N_1)) * A_1^{-1} \\ &= C_2 * A_1^{-1} + p_1 * (M_1, N_1) * A_1^{-1} \end{aligned}$$

Authorized licensed use limited to: Foshan University. Downloaded on September 06,2022 at 13:38:10 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

According to the Lemma 1, since

$$C_1 = (x_1, y_1)$$
$$C_2 = C_1 * A_1 \% det(A_1)$$

then,

$$C_2 * A_1^{-1} = C_1 + (M_2, N_2), M_2, N_2 \in \mathbb{Z}$$

On the other hand,

$$\begin{aligned} p_1 * (M_1, N_1) * A_1^{-1} &= p_1 * (M_1, N_1) * \frac{1}{p_1} * \begin{bmatrix} d_1 & -b_1 \\ -c_1 & a_1 \end{bmatrix} \\ &= (M_1, N_1) * \begin{bmatrix} d_1 & -b_1 \\ -c_1 & a_1 \end{bmatrix} \\ &= (M_3, N_3), M_3, N_3 \in \mathbb{Z} \end{aligned}$$

Thus,

$$\begin{aligned} C_5 &= C_2 * A_1^{-1} + p_1 * (M_1, N_1) * A_1^{-1} \\ &= C_1 + (M_2, N_2) + (M_3, N_3) \\ &= C_1 + (M, N), M, N \in \mathbb{Z} \end{aligned}$$

□

Extend the Proposition 1 to the case where there are n switches and we have the following proposition.

Proposition 2: For ith switch $S_i (i = 1, 2, 3 \ldots n)$ whose matrix is

$$A_i = \begin{bmatrix} a_i & b_i \\ c_i & d_i \end{bmatrix},$$

if

$$\begin{aligned} C_1 &= (x_1, y_1), x_1, y_1 \in \mathbb{Z} \\ C_j &= C_{j-1} * A_{j-1} \% p_{j-1} (2 \le j \le n+1). \end{aligned}$$

Then, by setting

$$p_i = det(A_1) * det(A_2) * \ldots * det(A_i),$$

It can be proved that

$$D_{n+1} = C_{n+1} * A_n^{-1} * A_{n-1}^{-1} * \ldots * A_1^{-1}.$$

consists of integers.

Proof: We apply mathematical induction to prove this proposition. When there are 2 switches, this proposition degenerates to Proposition 1, which has been proved to be correct.

Assume the proposition holds when there are $k (3 \le k \le n-1)$ switches, namely:

$$D_{k+1} = C_{k+1} * A_k^{-1} * A_{k-1}^{-1} * \ldots * A_1^{-1}$$

consists of integers. Then, when there are $(k+1)$ switches, we need to prove that:

$$D_{k+2} = C_{k+2} * A_{k+1}^{-1} * A_k^{-1} * \ldots * A_1^{-1}$$

also consists of integers. Since

$$\begin{aligned} C_{k+2} &= C_{k+1} * A_{k+1} \% p_{k+1} \\ &= C_{k+1} * A_{k+1} \% (det(A_1) * det(A_2) * \ldots * det(A_{k+1})) \end{aligned}$$

According to the Lemma 1, we have:

$$\begin{aligned} C_{k+2} * A_{k+1}^{-1} &= C_{k+1} + q * (M_1, N_1) \quad M_1, N_1 \in \mathbb{Z} \\ where \quad q &= det(A_1) * det(A_2) * \ldots * det(A_k) \end{aligned}$$

Thus

$$\begin{aligned} D_{k+2} &= C_{k+2} * A_{k+1}^{-1} * A_k^{-1} * \ldots * A_1^{-1} \\ &= (C_{k+1} + q * (M_1, N_1)) * A_k^{-1} * A_{k-1}^{-1} * \ldots * A_1^{-1} \\ &= D_{k+1} + q * (M_1, N_1) * A_k^{-1} * A_{k-1}^{-1} * \ldots * A_1^{-1} \\ &= D_{k+1} + (M, N). \quad M, N \in \mathbb{Z}. \end{aligned}$$

Since D_{k+1} consists of integers, therefore, D_{k+2} also consists of integers and the proposition is proved. □

Turn back to our **Problem 1**, it is evident that we could set $p_i = det(M_1) * det(M_2) * \ldots * det(M_i) (i = 1, 2, 3 \ldots, n)$, so that $(v_{2i+1}, v_{2i+2}) * M_i^{-1}$ consists of integers.

B. Matrix Design

Because we use the inverse of the matrix to recover the actual path and localize the faulty switch, we must guarantee that the inverse of the matrix exists. Suppose there is a $2 * 2$ matrix $M = [\vec{a}_1, \vec{a}_2]$, then, M is invertible if and only if $\vec{a}_1$ and $\vec{a}_2$ are linearly independent, which is true if and only if neither of them is a multiple of the other.

Besides, if we set $p_i = det(M_1) * det(M_2) * \ldots * det(M_i) (i = 1, 2, 3 \ldots, n)$, then p_i may overflow, except that we restrict all determinants to a small value, such as 2 or 3. The larger determinants are, the smaller network scale that 2MVeri could be applied to. If we set all determinants to 2 and use a 32-bit field in ip header to store the modulus, the longest path in network could contain 31 switches or so. If we set all determinants to 4 and use a 32-bit field in ip header to store the modulus, the longest path in network could only contain 15 switches or so. Evaluations in the next section show that when all determinants are set to 2, the accuracy of verification and localization is unacceptable, while with determinants larger than 2, 2MVeri could achieve almost the same performance as no modular mechanism is applied when there is no overflow.

To summary, the matrix should satisfy the following constraints:

1. Neither of the column vectors in the matrix is a multiple of the other.
2. The determinants should be larger than 2 and to avoid overflow, they should be limited to an upper bound, which varies according to the network scale.

C. Vector Design

Consider a special case where the two initial values of the vector are both multiples of the determinant of the matrix of the entry switch, the vector will be zero after passing the entry switch, in which situation 2MVeri is unable to recover the actual path. Thus, the entry switch should be responsible for not assigning multiples of the determinant of its matrix as the initial value of vectors.

VI. EXPERIMENTS

A. Experiment Setup

We emulate networks with Mininet, consisting of Open vSwitch (OVS) [25] instances, and use Floodlight as the controller. The verification server runs on a desktop with

Authorized licensed use limited to: Foshan University. Downloaded on September 06,2022 at 13:38:10 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

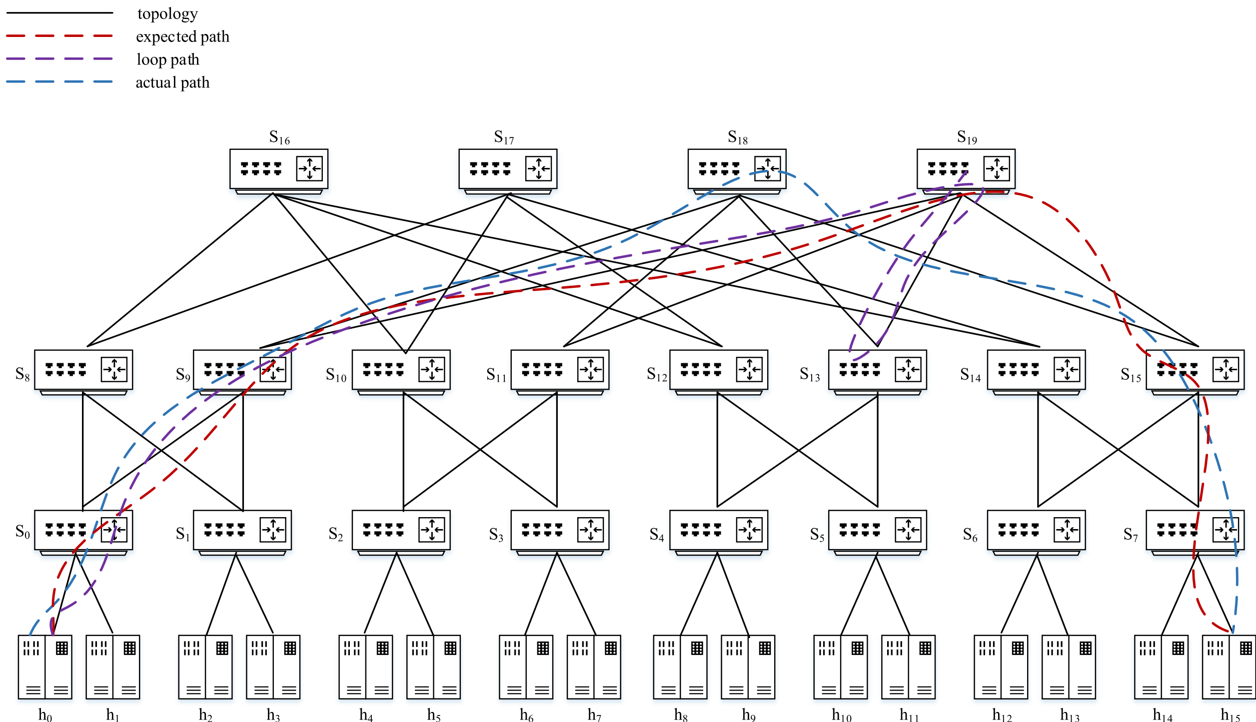

Fig. 6. Fat tree topology with k = 4 is used in the functional test.

an Inter Core i7 3.6GHz and 16GB Memory. We use the following three topologies for experiments.

1) Topology: We let the emulated hosts ping each other in order to populate the switches' flow table with shortest-path forwarding rules. The Internet2 topology consists of 9 Juniper routers. We use the 112,873 IPv4 forwarding rules. We emulate fat tree (both k=4 and k=6) topologies which represent medium-sized networks. The Standford backbone network [26] consist of 16 Cisco routers and 10 layer-2 switches. There are 710,236 forwarding rules. For Standford and Internet2, we translate the configuration files to equivalent OpenFlow rules, and install them at Open vSwitch wich Floodlight. We remove all the loops in the Standford and Internet2 topologies.

2) Bloom Filter: We use the approach described in [15]. First, we construct three hash functions: $g_0(x), g_1(x), g_2(x)$, where $g_i(x) = h_1(x) + ih_2(x)(i = 0, 1, 2)$. $h_1(x)$ and $h_2(x)$ are the two halves of a 32-bit Murmur3 hash of x [27]. Then, we use the first 4 bits of $g_i(x)$ to set the 16-bit Bloom filter.

B. Function Test

For function test, we generate faults in the fat tree k=4 topology, and test whether 2MVeri can detect these faults and pinpoint the faulty switches.

1) Black Hole: In Fig. 6, there is a flow from host h_0 to h_{15} with destination address 10.0.0.16/24. The expected path for these packets is $h_0 \rightarrow S_0 \rightarrow S_9 \rightarrow S_{19} \rightarrow S_{15} \rightarrow S_7 \rightarrow h_{15}$ (red solid line). we create a fault by modifying the action of the forwarding rule in switch S_9 that matched dst ip = 10.0.0.16/24 with a drop action. Then, the flow will be dropped at S_9. In this case, 2MVeri immediately detects the fault, and localizes the faulty switch S_9.

2) Path Deviation: We continue to create a fault that causes the previous flow to deviate from the original path, by modifying the same rule. This time, we replace the action to forward towards switch S_{18}. The expected path for these packets is $h_0 \rightarrow S_0 \rightarrow S_9 \rightarrow S_{19} \rightarrow S_{15} \rightarrow S_7 \rightarrow h_{15}$ (red solid line). But the actual path is $h_0 \rightarrow S_0 \rightarrow S_9 \rightarrow S_{18} \rightarrow S_{15} \rightarrow S_7 \rightarrow h_{15}$ (blue dot line). 2MVeri immediately detects the fault, and recover the actual path, thereby localizing the faulty switch S_9.

3) Loop: We create a loop by modifying the action of the forwarding rule in switch S_{19} that matched dst ip = 10.0.0.16/24, from the original forwarding to switch S_{15} to forwarding to switch S_{13}. Packets are continuously forwarded between S_{19} and S_{13}, resulting in packet loss. In this case, the reported tag is inconsistent with the correct tag, resulting in verification failure. 2Mveri immediately detects the fault and localizing the faulty switch S_{19}.

C. Verification Accuracy

We let hosts send udp traffic to each other to set up correct flow tables in each switch. After the flow tables have been established, we randomly choose some host pairs. For each chosen host pair, we randomly select a switch in the forwarding path of the udp traffic and change its flow tables so that the packets are not forwarded properly, resulting in the inconsistency between the control plane and the data plane.

The verification of 2MVeri has no false positives. The reason is that if the packet takes the expected path, the reported tag will be the same as the correct tag, and the verification will

Authorized licensed use limited to: Foshan University. Downloaded on September 06,2022 at 13:38:10 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

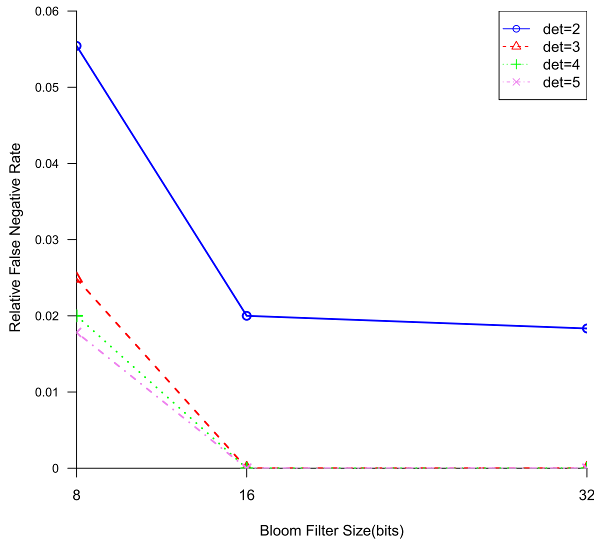

Fig. 7. Relative false negative rate of 2Mveri in fat tree with k=4.

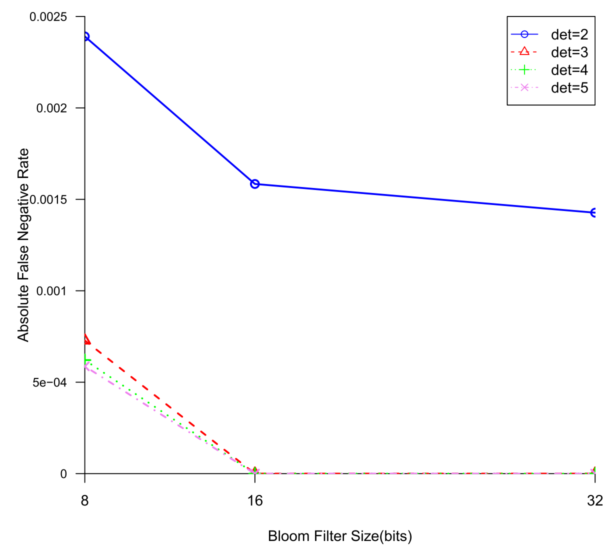

Fig. 8. Absolute false negative rate of 2Mveri in fat tree with k=4.

succeed. False negatives happen if and only if the reported tag and the correct tag are identical while the packet has taken a different path from the expected one.

To evaluate the verification accuracy, we use absolute false negative rate and relative false negative rate described in [15] as metrics and compare the result of 2MVeri with VeriDP [15]. Let n be the number of total packets we generate after simulating faults, $n1$ be the number of packets that arrive at the destination host, and $n2$ be the number of packets that arrive at the destination host with the same tags as the correct ones. The absolute false negative rate is defined as $n2/n$, and the relative false negative rate is defined as $n2/n1$.

First, we evaluate the effect of modulus value on the verification accuracy. 2MVeri adjusts and enlarges the value of modulus p based on cumulative multiplication of characteristic matrix determinant. Therefore, Evaluating the effect of modulus value is equivalent to evaluate the effect of characteristic matrix determinant. In fat tree both $k = 4$ and $k = 6$ topologies, we set the characteristic matrix determinants of all switch to 2,3,4,5, respectively. Fig. 7 and Fig. 8 show the effect of different modulus values on 2Mveri path verification accuracy in fat tree k=4 topology, Fig. 9 and Fig. 10 is in fat tree k=6 topology.

As shown in Fig.7 and Fig. 8, in our experiment, regardless of Bloom filter size, the smaller the matrix determinant is, the higher the false negative rate of path verification of 2MVeri. Especially when the determinant is set to 2, the false negative rate of path verification is the highest. When the determinant is set to 3,4,5, the false negative rate of path validation does not change much.

As shown in Fig.9 and Fig. 10, although the false negative rate of path validation was the highest when the matrix determinant is set to 2, the false negative rate gradually increased as the determinant changed from 3 to 4 and finally to 5. This is because although the increment of matrix determinant is very small, the value of modulus p is based on cumulative multiplication of characteristic matrix determinant on the

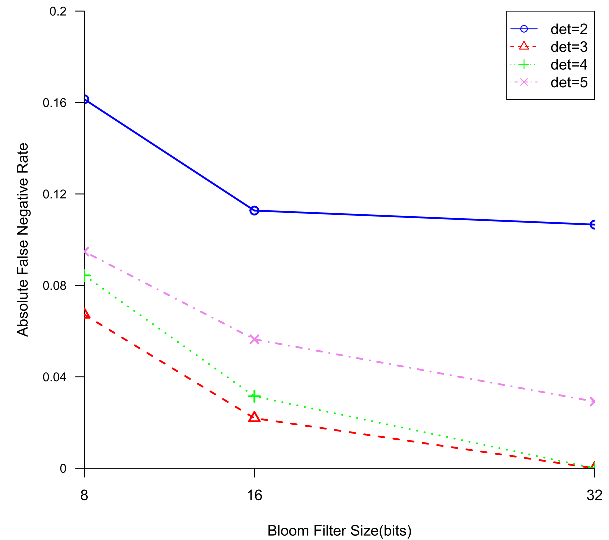

Fig. 9. Absolute false negative rate of 2Mveri in fat tree k=6.

actual path. Therefore, in the process of increasing the matrix determinant, the modulus is also increasing, and the growth rate is very fast. Although the increase of modulus reduces the loss of path information in the process of tag updating, the increase of modulus produces overflow, which leads to the decrease of the accuracy of path verification.

According to the above results, when the characteristic matrix determinant is set to 3, the accuracy of 2MVeri path verification is the highest.

D. Localization Accuracy

We collect the reported tags which fail the verification and try to recover the actual paths of these packets. The controller uses the localization algorithm to try to recover the actual path that the packet passed through and locate to the faulty switch. Let n be the number of packets that led to the failure of the path verification, and m be the number of packets that

Authorized licensed use limited to: Foshan University. Downloaded on September 06,2022 at 13:38:10 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

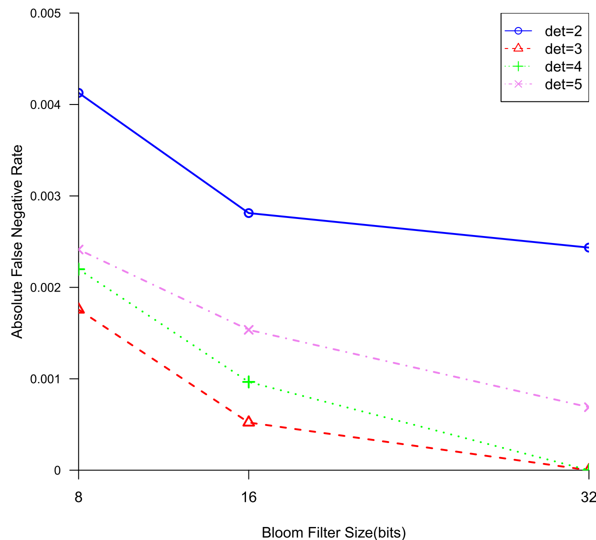

Fig. 10. Absolute false negative rate of 2Mveri in fat tree k=6.

have recovered the correct actual path, then m/n represents the localization accuracy rate.

First, we evaluate the effect of modulus value on the localization accuracy. 2MVeri adjusts and enlarges the value of modulus p based on cumulative multiplication of characteristic matrix determinant. Therefore, Evaluating the effect of modulus value is equivalent to evaluate the effect of characteristic matrix determinant. In fat tree both $k = 4$ and $k = 6$ topologies, we set the characteristic matrix determinants of all switch to 2,3,4,5, respectively.

The results in Table II show that when the determinant of the characteristic matrix is smaller, the localization accuracy of 2Mveri is lower, especially when determinants are set to 2, localization accuracy of 2Mveri is the lowest. When determinants are set to 3,4,5, localization accuracy does not change much.

In 2Mveri, the value of modulus is positively related to matrix determinant. The larger the value of determinant is, the larger the value of modulus will be. In the process of label updating, the less path information lost due to modular operation will be, so localization accuracy will be improved.

However, the above conclusion is only true if the modulus does not cause overflow. As the network scale increases and the modulus increases, localization accuracy decreases. Table III shows the experimental results in the fat tree k = 6.

Although localization accuracy is still the lowest when all determinants are set to 2, localization accuracy gradually decreased as determinants changed from 3 to 4 and finally to 5. This is because although the increment of matrix determinant is very small, the value of modulus p is based on cumulative multiplication of determinant on the actual path. Therefore, in the process of increasing the matrix determinant, the modulus is also increasing, and the growth rate is very fast. Although the increase of modulus reduces the loss of path information in the process of tag updating, the increase of modulus produces overflow, which leads to the decrease of localization accuracy.

According to II and Table III, when the characteristic matrix determinant is set to 3, the localization accuracy is the highest.

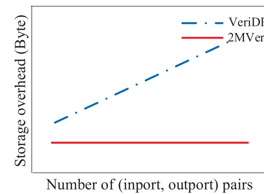

Fig. 11. Storage overhead (1).

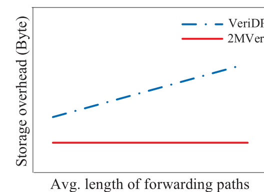

Fig. 12. Storage overhead (2).

so in the subsequent evaluation, all matrix determinants are set to 3.

Table IV shows the localization accuracy of VeriDP and 2Mveri in the fat tree k = 4 and k = 6. Due to the increasing complexity of network connection, localization accuracy in k = 6 is lower than that in k = 4. In the same network topology, the localization accuracy of 2Mveri is the highest.

E. Control Plane Overhead

We analyze the control plane overhead of 2MVeri and VeriDP [15] theoretically from the following two aspects.

1) Storage Overhead: In the control plane, VeriDP needs to store an additional path table where each $(inport, outport)$ pair corresponds to (1) a list of packet headers which belong to a series of flows that enter the network at $inport$ and leave at $outport$, (2) a set of hop-by-hop forwarding paths in the form of $(input_port, switch_id, output_port)$, (3) a number of 16-bit Bloom filters. Obviously, the storage overhead of the path table is proportional to the number of $(inport, outport)$ pairs and the average length of forwarding paths. While the control plane storage overhead of 2MVeri will be a constant, as long as the number of flows and switches keep unchanged.

As illustrated in Fig. 11 and Fig. 12, if the number of $(inport, outport)$ pairs increases, the number of entries in the path table will increase, causing larger overhead. As for the case where the average length of forwarding paths increases, although the number of path table entries remains the same, the size of each entry increases, introducing larger overhead as well. However, in 2MVeri, the storage overhead is irrelevant to the number of port pairs or the length of forwarding paths.

2) Computational Overhead: The computational overhead of VeriDP comes from three aspects: (1) path table construction, (2) verification, (3) localization, while the overhead of 2MVeri only comes from (2) and (3). To evaluate the overhead, we use *the number of pushing and popping operation* as the metric and a set of parameters of the network shown in Table V.

Table VI presents the computational overhead of VeriDP and 2MVeri in the worst case. Since the path table stores expected paths hop-by-hop for every flow from each input and output

Authorized licensed use limited to: Foshan University. Downloaded on September 06,2022 at 13:38:10 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE II
INFLUENCE OF MODULUS ON THE ERROR LOCATION ACCURACY OF 2MVERI IN FAT TREE WITH K=4

The value of matrix determinant	Number of failed verification	The number of packets recovered from the actual path	Localization Accuracy
Det=2	3927	3540	90.15%
Det=3	4087	4078	99.79%
Det=4	3642	3617	99.83%
Det=5	4127	4122	99.89%

TABLE III
INFLUENCE OF MODULUS ON THE ERROR LOCATION ACCURACY OF 2MVERI IN FAT TREE WITH K=6

The value of matrix determinant	Number of failed verification	The number of packets recovered from the actual path	Localization Accuracy
Det=2	8402	7261	86.43%
Det=3	8932	8764	98.12%
Det=4	8736	8511	97.43%
Det=5	8674	8294	95.62%

TABLE IV
INFLUENCE OF MODULUS ON THE ERROR LOCATION ACCURACY OF 2MVERI IN FAT TREE

The value of matrix determinant	Number of failed verification	The number of packets recovered from the actual path	Localization Accuracy
VeriDP (k=4)	2527	2505	99.2%
2Mveri (k=4)	3857	3848	99.79%
VeriDP (k=6)	7148	6902	96.6%
2Mveri (k=6)	7968	7800	97.79%

TABLE V
PARAMETERS OF THE NETWORK

Parameters	Meanings
n	The number of $(inport, outport)$ pairs.
m	The number of flows in the network.
p	The maximum number of ports in each switch.
h	The maximum number of hops in forwarding paths.

TABLE VI
COMPUTATIONAL OVERHEAD OF VERIDP AND 2MVERI

Schemes	Path table	Verification	Localization
VeriDP	$O(nmh)$	$O(n)+O(m)$	$O(ph) * (O(n)+O(m)+O(h))$
2MVeri	0	$O(1)$	$O(ph)$

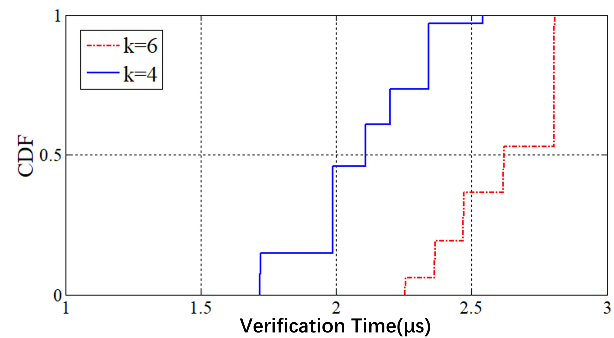

Fig. 13. Time to verify a tag report on 2MVeri (fat tree topology with k=4 and k=6).

port pair, the overhead to construct the path table is $O(nmh)$. For verification, VeriDP uses the reported $(inport, outport)$ pair and headers to match on the path table, which causes $O(n)+O(m)$ overhead per packet. While 2MVeri only needs to compare the reported tag with the correct one whose overhead is $O(1)$ per packet (the correct tag only needs to be computed once for each flow and the overhead is $O(h)$). For localization, VeriDP recovers the actual path in two phases: (1) constructing the common part with the expected path, (2) constructing the different part based on backtrace. The construction and backtrace result in $O(ph)*(O(n)+O(m)+O(h))$ overhead. As for 2MVeri, it needs to traverse all the possible switches to find the faulty one, causing $O(ph)$ overhead.

We also use the method(Section VI, A, Topology) so that the data packets are not forwarded according to the originally determined correct path, simulating the situation that the path is wrong. After a network error, this section collects all the reported tags, runs the path verification algorithm 10^4times for each reported tag, and records the time it takes to run the path verification algorithm every 10^4times. The average of the time spent by the four path verification algorithms is the time for one path verification for a single reported tag. The Fig.13 shows the Cumulative Distribution Function (CDF) of the time it takes to perform a path verification on a single reported tag.

In Fig.13, the time it takes for 2MVeri to perform a path verification on a single tag is concentrated between 1.7μs and 2.8μs, which means that the throughput of 2MVeri for path verification can be up to 5.88×10^5 times per second, and at least it can reach 3.57×10^5 times per second. In contrast, the time spent by VeriDP path verification for a single tag is between 2μs and 3μs, which means that the throughput of VeriDP's path verification is only 5.00×10^5 times per second at most. At least it is only 3.33×10^5 times per second.

VII. CONCLUSION AND FUTURE WORK

This paper proposes 2MVeri, a new framework that exploits the path information in the tag, which consists of a two-dimensional vector, a modulus and a Bloom filter, to measure the control-data plane consistency. 2MVeri uses a modules, a Bloom filter and a two-dimensional vector as a tag which is inserted in the packet header and is updated in each switch

Authorized licensed use limited to: Foshan University. Downloaded on September 06,2022 at 13:38:10 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

that the packet traverses. By exploiting path information compressed in the tag, 2MVeri can verify the consistency between the data and control plane. Moreover, when verification fails, 2MVeri is able to localize the faulty switch. Experimental results show that in the k = 4 fat tree topology, the verification accuracy of 2MVeri is as high as 100%. In addition, when the actual path is inconsistent with the expected path, 2MVeri can locate the wrong switch with an accuracy of 99.8%. This is also inconsistent performance, and we will conduct related research in the next stage of work if the expected and actual paths are same but rules matched on the switch are different. This paper hopes that the 2MVeri scheme is not limited to the ipv4 protocol scenario, but can be extended to the scenario where multiple protocols coexist, so that it can adapt to the newly proposed protocol. At the same time, considering the flexibility, programmability, reusability and other advantages of P4 software switch [28], [29] in multi-protocol coexistence scenarios,this paper hopes to implement the 2Mveri framework on P4 software switch in the future, and then extend it to the scenario independent of data plane protocol.

REFERENCES

[1] D. Kreutz, F. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-defined networking: A comprehensive survey,” *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.

[2] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, “A survey of software-defined networking: Past, present, and future of programmable networks,” *IEEE Commun. Surveys Tuts.*, vol. 16, no. 3, pp. 1617–1634, 3rd Quart., 2014.

[3] H. Kim and N. Feamster, “Improving network management with software defined networking,” *IEEE Commun. Mag.*, vol. 51, no. 2, pp. 114–119, Feb. 2013.

[4] *Openflow Switch Specification Version 1.5.1*. [Online]. Available: https://3vf60mmveq1g8vzn48q2o71a-wpengine.netdna-ssl.com/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

[5] K. Bu, X. Wen, B. Yang, Y. Chen, L. E. Li, and X. Chen, “Is every flow on the right track?: Inspect SDN forwarding with RuleScope,” in *Proc. IEEE INFOCOM 35th Annu. Int. Conf. Comput. Commun.*, Apr. 2016, pp. 1–9.

[6] M. Kuźniar, P. Perešíni, and D. Kostić, “What you need to know about SDN flow tables,” in *Proc. Int. Conf. Passive Act. Netw. Meas.* Cham, Switzerland: Springer, 2017, pp. 347–359.

[7] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, “OFLOPS: An open framework for OpenFlow switch evaluation,” in *Passive and Active Measurement* (Lecture Notes in Computer Science), vol. 7192. Berlin, Germany: Springer, 2012, pp. 85–95.

[8] M. Kuzniar, P. Peresini, and D. Kostić, “Providing reliable FIB update acknowledgments in SDN,” in *Proc. 10th ACM Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2014, pp. 415–422.

[9] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, “Automatic test packet generation,” in *Proc. 8th Int. Conf. Emerg. Netw. Experiments Technol. (CoNEXT)*, 2012, pp. 241–252.

[10] *Open Network Install Environment (Onie)*. [Online]. Available: http://onie.org/

[11] S. Hong, L. Xu, H. Wang, and G. Gu, “Poisoning network visibility in software-defined networks: New attacks and countermeasures,” in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–15.

[12] R. Mahajan and R. Wattenhofer, “On consistent updates in software defined networks,” in *Proc. 12th ACM Workshop Hot Topics Netw.*, Nov. 2013, p. 20.

[13] A. Shukla, S. Schmid, A. Feldmann, A. Ludwig, S. Dudycz, and A. Schuetze, “Towards transiently secure updates in asynchronous SDNs,” in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 597–598.

[14] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, “Abstractions for network update,” in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit., Protocols Comput. Commun. (SIGCOMM)*, 2012, pp. 323–334.

[15] P. Zhang *et al.*, “Mind the gap: Monitoring the control-data plane consistency in software defined networks,” in *Proc. 12th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2016, pp. 19–33.

[16] P. Zhang, “Towards rule enforcement verification for software defined networks,” in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, May 2017, pp. 1–9.

[17] V. K. S. Dara, S. S. Bhandari, A. Yourtchenko, E. Vyncke, and F. Brockners, “Network path proof of transit using in-band metadata,” U.S. Patent 14 992 109, Oct. 27, 2018.

[18] P. Perešíni, M. Kuźniar, and D. Kostić, “Monocle: Dynamic, fine-grained data plane monitoring,” in *Proc. 11th ACM Conf. Emerg. Netw. Exp. Technol.*, Dec. 2015, p. 32.

[19] M. Kuzniar, P. Peresini, and D. Kostic, “Proboscope: Data plane probe packet generation,” Tech. Rep., 2014.

[20] K. Lei, K. Li, J. Huang, W. Li, J. Xing, and Y. Wang, “Measuring the control-data plane consistency in software defined networking,” in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2018, pp. 1–7.

[21] *Shamir’s Secret Sharing*. [Online]. Available: https://en.wikipedia.org/wiki/Shamir’s_Secret_Sharing

[22] A. Shaghaghi, M. A. Kaafar, and S. Jha, “WedgeTail: An intrusion prevention system for the data plane of software defined networks,” in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, Apr. 2017, pp. 849–861.

[23] A. Shukla, S. J. Saidi, S. Schmid, M. Canini, T. Zinner, and A. Feldmann, “Toward consistent SDNs: A case for network state fuzzing,” *IEEE Trans. Netw. Service Manage.*, vol. 17, no. 2, pp. 668–681, Jun. 2020.

[24] A. Shukla, S. Fathalli, T. Zinner, A. Hecker, and S. Schmid, “P4Consist: Toward consistent p4 SDNs,” *IEEE J. Sel. Areas Commun.*, vol. 38, no. 7, pp. 1293–1307, Jul. 2020.

[25] *Openvswitch*. [Online]. Available: http://openvswitch.org/

[26] P. Kazemian, G. Varghese, and N. Mckeown, “Header space analysis: Static checking for networks,” in *Proc. Usenix Conf. Netw. Syst. Design Implement.*, 2012, pp. 113–126.

[27] *The Murmur3 Hash Function*. [Online]. Available: https://github.com/aappleby/smhasher

[28] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, and J. Rexford, “Pisces: A programmable, protocol-independent software switch,” in *Proc. ACM Sigcomm Conf.*, 2016, pp. 525–538.

[29] P. Bosshart *et al.*, “P4: Programming protocol-independent packet processors,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

Authorized licensed use limited to: Foshan University. Downloaded on September 06,2022 at 13:38:10 UTC from IEEE Xplore. Restrictions apply.