Verifying Rule Enforcement in Software Defined Networks With REV

Peng Zhang, *Member, IEEE*, Hui Wu, Dan Zhang, and Qi Li, *Senior Member, IEEE*

***Abstract*— Software defined networking (SDN) reshapes the ossified network architectures, by decoupling the control plane and data plane. Due to such a decoupling, SDN assumes that rules issued by the control plane are always correctly enforced by the data plane. However, this assumption breaks as an adversary can prevent the data plane from enforcing the rules, by exploiting the vulnerabilities of switch OS and control channel. The serious consequence is that packets may deviate from their original paths, thereby violating critical security policies like access control. To this end, this paper introduces *rule enforcement verification (REV)*, which enables the controller to check whether switches have correctly enforced the rules that it issues. Since using message authentication code (MAC) can incur heavy switch-to-controller traffic, we propose the *compressive MAC*, which lets switches compress MACs before reporting to the controller, thereby significantly reducing the bandwidth cost. Finally, we propose a heuristic flow selection algorithm, which allows the controller to verify much less flows for rule coverage. We implement REV based on Open vSwitch with DPDK, and use experiments to show: (1) by using compressive MAC, REV achieves a 97% reduction in switch-to-controller traffic, and an 8× increase in verification throughput; (2) by using the heuristic flow selection algorithm, REV can reduce the number of flows to verify by 40%-60%.**

***Index Terms*— Software-defined networks, rule enforcement verification, compressive MAC.**

I. Introduction

SOFTWARE Defined Networking (SDN) promises a centralized, flexible, and programmable control of computer networks. At the core of SDN is the decoupling of control plane and data plane: in the control plane, a controller compiles network policies (e.g., routing, access control, waypoint traversal) into rules, and installs them at switches through a standard control channel (e.g., OpenFlow [2]); in the data plane, switches enforce these rules to realize the network policies.

Even the decoupling of control plane and data plane offers SDN many benefits, it also raises new security concerns due to the following two vulnerabilities.

(1) SDN switches are vulnerable to compromises. We already witnessed a lot of switch compromises in traditional networks [3]–[6], and SDN switches running third-party OSes tend to more vulnerable and can result in even severe compromises [7]–[12]. For example, Pickett shows that an attacker can compromise the boot loader of switch OSes, so as to persistently control SDN switches [7].

(2) The control channel lacks security protection. After OpenFlow 1.0, SSL/TLS becomes optional rather than mandatory [13]. Due to the complexity of deploying SSL/TLS, many SDN switch and controllers just forgo this feature in the early days [14]. Even more recent controllers like ONOS [15] and Ryu [16] support SSL/TLS, many switch OSes still lack support of SSL/TLS or use outdated libraries [7]. Finally, SSL/TLS has its own vulnerabilities and cannot eliminate the possibility of man-in-the-middle attacks [17], [18].

The above vulnerabilities can be exploited by an adversary to prevent switches from correctly enforcing rules issued by the controller. As a result, packets may deviate from their original paths, violating critical network policies. For example, by deleting a rule that directs flows towards a firewall, the adversary can make these flows bypass the firewall traversal policy.

Even there are many network verification tools, most of them aim to check the correctness of network policies [19]–[22], or check the enforcement of policies while assuming switches are trustable [23]–[27]. As a result, they cannot check whether switches enforce network policies in an adversarial setting. SDNsec [28] uses message authentication code (MAC) to validate whether the path of a flow is consistent with network policies. However, SDNsec can place large bandwidth overhead on the control channel when verifying a flow of many packets.

To this end, this paper proposes *Rule Enforcement Verification (REV)*, a new security primitive for SDN. At a high level, REV allows a controller to *securely* check whether rules issued by it have been correctly enforced by switches. Similar to SDNsec, we realize REV based on message authentication code (MAC). When a packet enters the network, the entry switch attaches a tag to the packet. Then, each downstream switch updates the tag with the secret key shared with the controller. When the tagged packet is about to leave the network,

Manuscript received December 25, 2018; revised November 7, 2019; accepted February 19, 2020; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor W. Lou. Date of publication March 12, 2020; date of current version April 16, 2020. This work was supported in part by the National Key Research and Development Program of China under Grant 2017YFB0801703, in part by the National Natural Science Foundation of China under Grant 61772412, Grant 61572278, and Grant U1736209, and in part by the BNRist Network and Software Security Research Program under Grant BNR2019TD01004. The work of Peng Zhang was supported by the K. C. Wong Education Foundation. A preliminary version of this article has been published in the Proceedings of IEEE INFOCOM 2017. *(Corresponding author: Qi Li.)*

Peng Zhang, Hui Wu, and Dan Zhang are with the School of Computer Science and Technology, MOE Key Laboratory for Intelligent Networks and Network Security, Xi'an Jiaotong University, Xi'an 710049, China (e-mail: p-zhang@xjtu.edu.cn).

Qi Li is with the Institute of Cyber Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, and also with the Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, Beijing 100084, China (e-mail: qli01@tsinghua.edu.cn).

This article has supplementary downloadable material available at http://ieeexplore.ieee.org, provided by the authors.

Digital Object Identifier 10.1109/TNET.2020.2977006

it is reported to the controller by the exit switch. Finally, the controller verifies the packet against its tag, in order to determine whether the packet has traversed the intended path according to the rules.

Using standard MACs, both the entry and exit switch need to report each packet and its tag to the controller, which will result in a large volume of switch-to-controller traffic (a common limitation shared by SDNsec). To handle this challenge, this paper proposes *compressive message authentication code (compressive MAC)*, a novel MAC that enables edge switches to compress tags before reporting to the controller. Specifically, both the entry and exit switch combine packets (with tags) belonging to the same flow into a single *flow-packet*, and only report this flow-packet to the controller when the flow ends. Once the flow-packet passes the verification, it holds with high probability that each packet of the flow has traversed the intended path, or equivalently, enforced the rules. We prove the security of compressive MAC under standard cryptographic models.

Compared to the preliminary version of REV [1], this paper further extends REV to verify whether *all* active rules in the network have been correctly enforced. Instead of verifying every flow in the network, we use a heuristic flow selection algorithm that can cover the maximum number of rules with much less flows.

Finally, noted the performance limitation of our preliminary implementation on CPqD [29], in this paper we implement REV based on OVS-DPDK [30], i.e., Open vSwitch with DPDK data path, a high-performance software switch that can fast forward packets purely in user space. We show that REV based on OVS-DPDK can forward packets at near line rate.

In sum, our main contribution is three-fold:

- We propose rule enforcement verification (REV), a new security primitive for SDN, and implement it based on Open vSwitch with DPDK to achieve a near line rate forwarding.
- We propose a new message authentication code, named compressive MAC, and prove its security under standard cryptographic models. By using compressive MAC, we can achieve a 97% reduction in switch-to-controller traffic, and an $8\times$ improvement in verification throughput, compared with using standard MACs.
- We design a heuristic flow selection algorithm, and show it can reduce the number of flows to verify by 40%-60%, based on real network topologies.

The rest of this paper proceeds as follows. Section II states the problem of REV. Section III introduces design of REV. Section IV analyzes the security and accuracy of REV. Section V shows the implementation of REV, and Section VI evaluates its performance. Section VII surveys related work, Section VIII discusses remaining issues, and Section IX concludes.

II. Problem Statement

A. Network Model

This paper considers a typical software defined network consisting of one *controller* and multiple *switches*. The switches and their links collectively form the SDN *data plane*.

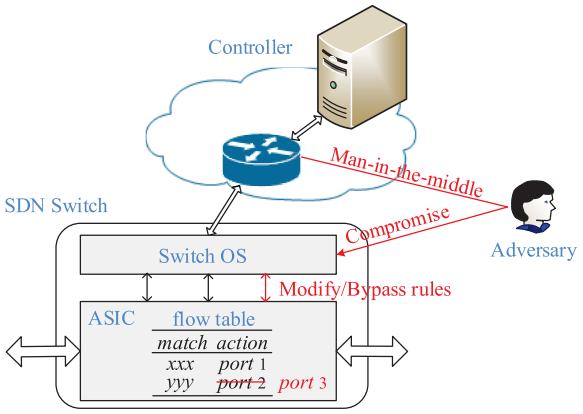


Fig. 1. Two ways for launching rule modification attack in SDN.

Network operators specify high-level policies such as "Host A should talk to Host B" with the API provided by the controller. The controller complies operators' policies into *rules*. A rule consists of two parts: *matching fields* and *actions*, saying that packets whose headers match the matching fields should take the corresponding actions. The controller populates the complied rules into *flow tables* of switches, through a standard *control channel* like OpenFlow. Switches forward packets by looking up in their flow tables. Define a *flow* as the set of packets with the same headers, e.g., all packets belonging to the same TCP session.

B. Threat Model

This paper considers what we call *rule modification attack*, where the adversary prevents switches from correctly enforcing the rules issued by the controller, by exploiting the following two vulnerabilities in SDN (as shown in Fig. 1). (1) The switch OS: by installing backdoor applications on the switch OS, the adversary can install, delete, modify rules installed at the switch, or simply bypass the rules and forward packets to ports not intended by the controller. (2) The control channel: by residing at intermediate switches as a man-in-the-middle, the adversary can inject, drop, or modify rule installation messages (e.g., flow-mod messages as in OpenFlow) sent by the controller.

Through rule modification attack, the adversary aims to change the forwarding behaviors of switches, thereby causing the following two forwarding anomalies.

Path Deviation. Packets take different paths than what are intended by the controller. To be more specific, path deviation can take the following three forms:

- **Switch Bypass.** One or more switches are skipped.
- **Path Detour.** The path deviates from one switch S_i to another switch other than the intended next-hop S_{i+1}, and comes back to S_{i+1} later.
- **Out-of-order Traversal.** Switches are not traversed in the order that they should appear on the forwarding path.

Unauthorized Access. Packets violate the access control policies and reach destinations that they are not authorized to. This can break isolation policies, e.g., only users from CS department can access a web server.

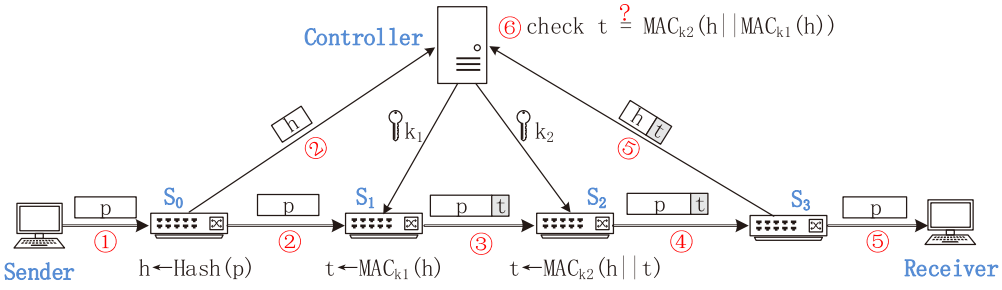


Fig. 2. An illustration of the packet-level REV method, where there is only one packet p to be verified. The sender sends p through a path $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3$. Before transmission, symmetric keys k_1 and k_2 are established between the controller and switches $S1$ and $S2$, respectively. Here, $Hash(\cdot)$ and $MAC_k(\cdot)$ are cryptographic hash function, and message authentication code with key k, respectively.

We assume multiple compromised switches can collude, while there should be at least one non-malicious switch along the deviated path, i.e., the expected forwarding path that are deviated from. For example, if S_i forwards packets directly to S_{i+n}, bypassing S_{i+1} through S_{i+n-1}, then the deviated path is $S_i \rightarrow S_{i+1} \rightarrow \cdots \rightarrow S_{i+n}$. As another example, if S_i forwards packets to other switches other than S_{i+1}, and the packets return to the correct path at S_{i+1}, then the deviated path is $S_i \rightarrow S_{i+1}$.

Compromised switches may try to disrupt the functionality of the controller, through other attacks like control plane saturation attack [31], topology poisoning attack [32], etc. Many excellent methods have been proposed to defend the controller against these attacks [31], [32], and REV can be used in combination with them.

C. Rule Enforcement Verification: Problem Definition

In the following, we formulate the problem of Rule Enforcement Verification (REV) in SDN. Let $\mathcal{F}$ be the set of all flows in the network. Consider a specific flow $f \in \mathcal{F}$, and let p be a packet of f. When p is received by the source switch S_0, it matches a set of rules (forwarding rules, ACL rules, etc.), which will collectively determine the next hop, say S_1. Then, p matches another set of rules at S_1, and be forwarded to S_2. This process continues until p reaches the destination switch S_{n+1}, which forwards p to the receiver.

Define $\mathcal{R}$ as the *network rule set*, i.e., the set of all rules at all switches of the network. Let $Path(\mathcal{R}, f)$ be the forwarding path of flow f, represented as $S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_{n+1}$. Let $R_f(S_i)$ be the set of rules matched by packets of f at switch S_i. Then, the set of all rules matched by flow f along its forwarding path can be defined as: $R_f = \cup_{S_i \in Path(\mathcal{R},f)} R_f(S_i)$. We give the definition of rule enforcement verification (REV) as follows.

Definition 1: **Packet-level REV** with respect to the network rule set $\mathcal{R}$, and a packet p of flow f is defined as: verifying whether p has traversed the path $Path(\mathcal{R}, f)$.

Definition 2: **Flow-level REV** with respect to the network rule set $\mathcal{R}$ and flow f is defined as: verifying whether all packets of flow f have traversed the path $Path(\mathcal{R}, f)$.

Definition 3: **Network-level REV** with respect to the network rule set $\mathcal{R}$ is defined as: selecting a subset $F \subset \mathcal{F}$ such that $\bigcup_{f \in F} R_f = \bigcup_{f \in \mathcal{F}} R_f$, and verifying whether all packets of each flow $f \in F$ have traversed the path $Path(\mathcal{R}, f)$.

Note that packet-level REV only verifies rule enforcement with respect to a single packet, while flow-level REV takes a flow of packets as a whole, and checks rule enforcement against all these packets. Finally, network-level REV ensures all active rules (those matched by at least one flow) in the network are enforced correctly by checking a collection of flows.

III. REV Methods

This section introduces three concrete methods to achieve packet-level REV, flow-level REV, and network-level REV, respectively. The relationship among these three methods is as follows. The packet-level REV method verifies whether a single packet has followed the rules issued by the controller. The flow-level REV method extends the packet-level REV method by compressing packets of a flow to reduce the bandwidth cost. Finally, the network-level REV method uses the flow-level REV method as a building block, and strategically selects flows to cover the maximum number of rules in the network.

A. Packet-Level REV

The basic idea of packet-level REV is illustrated in Fig. 2, where we are aimed to verify whether switches have correctly enforced rules to deliver a packet p. When the packet p enters the network, the source switch S_0 computes the hash value h of packet p, and sends h to the controller. The first-hop switch S_1 generates a tag t using its key k_1 shared with the controller. The second-hop switch S_2 updates the tag t using its key k_2 shared with the controller. When p is about to leave the network, the destination switch S_3 reports h and t to the controller. The controller checks whether t is the same with the tag computed by itself based on the rules.

Note in the above example, we have made some simplification on both switch processing and notations. The aim here is to convey the basic idea. In the following, we present the detailed procedure of packet-level REV, which has three stages: initialization, tagging, and verification.

1) Initialization: The packet-level REV method is organized in *sessions*. Each session verifies the rule enforcement with one packet, i.e., whether the packet has been forwarded according to the corresponding rules. In the following, let us consider one session, say v, and let f be the flow that the packet belongs to.

Assume the controller has a public/private key pair (pK, rK), and the controller shares a symmetric key K_i with each switch S_i in the network. The sharing of symmetric keys can be achieved using existing key establishment methods [33], [34]. This serves as a master key for generating session keys. In prior to a new REV session, the controller generates a *session identifier* as:

$$\texttt{VID} = H(\texttt{Inport}||\texttt{Header}||\texttt{TimeStamp}) \tag{1}$$

Here, H is a cryptographic hash function; `Inport` is the input port of f, e.g., a switchID/port pair; `Header` is the header of flow f, e.g., a wildcarded TCP five tuple; `TimeStamp` is the current time at the controller. Note that `Inport` and `Header` can be used to jointly specify the flow f. `TimeStamp` is used to prevent packet replays, where an adversary records and replays packets with out-of-date `VID`s.

Then, the controller sends a message containing `VID`, `Inport`, `Header`, `TimeStamp`, and a signature $\sigma_{rK}(\texttt{VID}||\texttt{TimeStamp})$ to the source switch through a secure channel. By secure, the channel should provide basic confidentiality and integrity for messages. The secure channel can be implemented with OpenFlow on top of SSL/TLS.

2) Tagging: When the source switch S_0 receives a packet p from port `Inport` and matching `Header`, it computes the hash of p as $\texttt{PktHash} = H(p)$, and sends a *verification report* containing `VID` and `PktHash` to the controller. Then, it sends p together with `PktHash`, `VID`, `TimeStamp`, and $\sigma_{rK}(\texttt{VID}||\texttt{TimeStamp})$ to the next hop S_1.

On receiving p, S_1 checks the validity of `VID` and `TimeStamp` against the signature, and derives its session key as $SK_1 \leftarrow F(K_1, \texttt{VID})$, where F is a pseudorandom function. Then, S_1 generates a tag t as:

$$t \leftarrow MAC_{SK_1}(\texttt{PktHash}) \tag{2}$$

Here, $MAC_k(\cdot)$ is the message authentication code with key k. Then, S_1 sends p together with t, `PktHash`, `VID`, `TimeStamp`, and $\sigma_{rK}(\texttt{VID}||\texttt{TimeStamp})$ to the next-hop S_2.

Each downstream switch S_i $(2 \leq i \leq n)$ conducts a similar process with S_1, and updates the tag t as:

$$t \leftarrow MAC_{SK_i}(\texttt{PktHash}||t) \tag{3}$$

When the destination switch S_{n+1} receives p, it sends p to the receiver. At the same time, S_{n+1} sends a verification report containing `VID`, `PktHash`, and t to the controller. Note that rather than storing the session keys, switches derive them from `VID` on demand. This approach is inspired by [35].

3) Verification: On receiving a verification report from a source switch, the controller saves it as a pending verification request. When the controller receives a verification report from a destination switch, with the same `VID` and `PktHash`, the controller checks whether the tag t and `PktHash` in the verification report satisfy the following equation:

$$t = MAC_{SK_n}(\texttt{PktHash}||MAC_{SK_{n-1}}(\texttt{PktHash}||\ldots \texttt{PktHash}||MAC_{SK_1}(\texttt{PktHash})\ldots)) \tag{4}$$

If so, the verification passes; otherwise, the verification fails.

B. Flow-Level REV

Even the packet-level REV method can verify the rule enforcement of a single packet, to verify a flow, one needs to apply the packet-level REV method for each packet of the flow. This means both the source and destination switch should send a verification report for each packet of the flow. When there are a large number of packets in a flow, the traffic of verification reports can cause congestion on the switch-to-controller channels, thereby preventing the controller from responding to normal switch requests (e.g., packet-in messages in OpenFlow).

This section introduces the flow-level REV method to overcome the above problem. In the flow level REV, the source/destination switch compresses packets of the same flow into a single *flow-packet*, and sends the flow-packet to the controller when the verification session ends. If the flow-packet passes the verification, then the controller can conclude that each packet of the flow has traversed the intended path.

The key challenge when realizing the flow-level REV is how to compress packets into a flow-packet, such that verifying a single flow-packet is equivalent to verifying all packets of the flow. This paper addresses this challenge by introducing *compressive MAC*, a new message authentication code that supports compression. To illustrate what a compressive MAC is, suppose there is a flow consisting of m packets $p_1, p_2, \ldots, p_m$, whose tags are $t_1, t_2, \ldots, t_m$, respectively. Informally, a compressive MAC should satisfy the following two conditions:

(1) $\texttt{Verify}(p_i, t_i) = \texttt{true}$, for $\forall i \in [1, m] \Rightarrow$
$\texttt{Verify}(\texttt{Compress}(\{p_i\}_{i=1}^m), \texttt{Compress}(\{t_i\}_{i=1}^m)) = \texttt{true}$
(2) $\texttt{Verify}(p_i, t_i) = \texttt{false}$, for $\exists i \in [1, m] \Rightarrow^{\text{(w.h.p)}}$
$\texttt{Verify}(\texttt{Compress}(\{p_i\}_{i=1}^m), \texttt{Compress}(\{t_i\}_{i=1}^m)) = \texttt{false}$

Here, `Verify` is the verification function, and `Compress` is the compression function. It is required that condition (1) should be always satisfied, and condition (2) should be satisfied with high probability (w.h.p.).

The construction of compressive MAC is inspired by homomorphic MACs [36], which in turn are based on the Cater-Wagman MAC [37]. Note that homomorphic MACs are designed for defending network coding against pollution attacks [38], while the compressive MAC has a quite different goal. In addition, the construction of compressive MAC is also quite different from those of homomorphic MACs.

Before introducing the compressive MAC, let us first define what a packet is: we represent a packet p by its hash value $\boldsymbol{p} = H(p)$. Here $\boldsymbol{p}$ is a vector of length l defined on a finite field $\mathbb{F}_q$, and $\boldsymbol{p}(i)$ refers to the ith element of $\boldsymbol{p}$.

The flow-level REV method consists of four stages: initialization, tagging, compression, and verification. Similar to the packet-level REV, let us consider a single verification session v for a flow f.

1) Initialization: This stage is similar to that of the packet-level REV method. First, the controller generates a session identifier `VID` using Eq. (1), and a compression key SK_c as $SK_c \leftarrow F(rk, \texttt{VID})$. Then, the controller sends a message containing `VID`, `Inport`, `Header`, `TimeStamp`, SK_c, and

the signatures $\sigma_{rK}(SK_c)$, $\sigma_{rK}(\texttt{VID}||\texttt{TimeStamp})$ to both the source and destination switches, through a secure channel as in the packet-level REV. The source and destination switches respectively create a *flow-packet* $\boldsymbol{p}_S^f \in \mathbb{F}_q^l$ and $\boldsymbol{p}_D^f \in \mathbb{F}_q^{l+n}$, both of which are initialized to all zeros.

2) Tagging: Before sending the ith packet $\boldsymbol{p}_i$, the source switch S_0 calculates its hash PktHash, and attaches $\boldsymbol{p}_i$ with VID, TimeStamp, PktHash, and the signature $\sigma_{rK}(\texttt{VID}||\texttt{TimeStamp})$. When the first-hop switch S_1 receives $\boldsymbol{p}_i$, it extracts the VID and generates session keys $SK_{1,a}$ and $SK_{1,b}$ using the master key K_1 shared with the controller as:

$$SK_{1,a} \leftarrow F(K_1, \texttt{VID}||0),\ SK_{1,b} \leftarrow F(K_1, \texttt{VID}||1)$$

After that, S_1 derives a vector $\boldsymbol{\alpha}_1 = G(SK_{1,a}) \in \mathbb{F}_q^l$, where G is a pseudorandom number generator (PRNG) with output on $\mathbb{F}_q^l$. Then, S_1 calculates a tag $t_{i,1}$ as:

$$\begin{aligned} a &\leftarrow \boldsymbol{p}_i \cdot \boldsymbol{\alpha}_1 = \sum_{j=1}^{l} \boldsymbol{p}_i(j)\boldsymbol{\alpha}_1(j) \in \mathbb{F}_q \\ b &\leftarrow F(SK_{1,b}, \texttt{VID}||S_0||i) \in \mathbb{F}_q \\ t_{i,1} &\leftarrow a + b \in \mathbb{F}_q, \end{aligned}$$

attaches $t_{i,1}$ at the end of $\boldsymbol{p}_i$, i.e., $\boldsymbol{p}_i \leftarrow \boldsymbol{p}_i||t_{i,1} \in \mathbb{F}_q^{l+1}$, and sends $\boldsymbol{p}_i$ to the next hop S_2.

Similarly, S_2 generates its session keys $SK_{2,a}$ and $SK_{2,b}$, derives the vector $\boldsymbol{\alpha}_2 = G(SK_{2,a}) \in \mathbb{F}_q^{l+1}$, and calculates a new tag $t_{i,2}$ as:

$$t_{i,2} \leftarrow \boldsymbol{p}_i \cdot \boldsymbol{\alpha}_2 + F(SK_{2,b}, \texttt{VID}||S_1||i) \in \mathbb{F}_q \tag{5}$$

Finally, S_2 sends $\boldsymbol{p}_i \leftarrow \boldsymbol{p}_i||t_{i,2} \in \mathbb{F}_q^{l+2}$ to S_3. This process continues until S_n sends packet $\boldsymbol{p}_i \in \mathbb{F}_q^{l+n}$ to the destination switch S_{n+1}. Note that each tag encodes the identity of the last-hop switch. This can prevent packet injection from out-of-path switches.

3) Compression: When the source switch sends the ith packet $\boldsymbol{p}_i$ (of length l), it derives $\beta_i \leftarrow F(SK_c, i) \in \mathbb{F}_q$, and updates its flow-packet $\boldsymbol{p}_S^f$ as:

$$\boldsymbol{p}_S^f(j) \leftarrow \boldsymbol{p}_S^f(j) + \beta_i \boldsymbol{p}_i(j),\ \text{for each } j \in [1, l]$$

When the destination switch receives the packet $\boldsymbol{p}_i$ (of length $l+n$), it similarly derives $\beta_i \leftarrow F(SK_c, i) \in \mathbb{F}_q$, and updates its flow-packet $\boldsymbol{p}_F^D$ as:

$$\boldsymbol{p}_D^f(j) \leftarrow \boldsymbol{p}_D^f(j) + \beta_i \boldsymbol{p}_i(j),\ \text{for each } j \in [1, l+n]$$

When the verification session ends, the source switch sends VID and $\boldsymbol{p}_S^f$ to the controller, and the destination switch sends VID and $\boldsymbol{p}_D^f$ to the controller.

4) Verification: The verification process at the controller is summarized in Algorithm 1. Here, the length of packet hash is l; the number of switches on the path (excluding the source and destination switches) is n; and the number of packets of the flow is m.

First, the controller determines the flow f according to VID, and computes the path of f based on the network rule set $\mathcal{R}$ (Line 1). Then the controller checks whether the following two conditions hold (Line 2-4): (1) the two flow-packets $\boldsymbol{p}_S^f$ and $\boldsymbol{p}_D^f$ carry the same packet hash; (2) the number of tags carried by $\boldsymbol{p}_D^f$ is n. Here, $\texttt{truncate}(\boldsymbol{p}_D^f, l)$ extracts the first l elements of vector $\boldsymbol{p}_D^f$. If either of the two conditions does not hold, the verification fails; otherwise, the controller checks all the n tags one by one (Line 5-13). For each switch i, the controller generates its session keys $SK_{i,a}$, $SK_{i,b}$, and computes $\boldsymbol{\alpha}$ (Line 6-7). Then, the controller calculates the inner product of $\boldsymbol{\alpha}$ and the first $l+i-1$ elements of p_D^f (Line 8), and the sum of all b's for those m packets (Line 9). If the sum of the above two values is equal to the ith tag in $\boldsymbol{p}_D^f$, then the verification passes for switch S_i and continues for S_{i+1}; otherwise the verification fails (Line 10-12). Finally, if the verification passes for all S_i's, the whole verification succeeds (Line 14).

Algorithm 1 $\texttt{Verify}(\texttt{VID}, f, m, \boldsymbol{p}_S^f, \boldsymbol{p}_D^f)$

Input: K_i: the private key of switch S_i,
$\mathcal{R}$: the network rule set,
VID: the verification session identifier,
f: the flow corresponding to VID,
m: the number of packets of flow f,
$\boldsymbol{p}_S^f$: the source flow-packet,
$\boldsymbol{p}_D^f$: the destination flow-packet.
Output: True or False.

1 Compute the forwarding path of f: $\texttt{Path}(\mathcal{R}, f) = (S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_{n+1})$;
2 **if** $\boldsymbol{p}_S^f \neq \texttt{truncate}(\boldsymbol{p}_D^f, l)$ *or* $\texttt{len}(\boldsymbol{p}_D^f) \neq n + l$ **then**
3 **return** False;
4 **end**
5 **foreach** $i \leftarrow 1$ **to** n **do**
6 $SK_{i,a} \leftarrow F(K_i, \texttt{VID}||0)$; $SK_{i,b} = F(K_i, \texttt{VID}||1)$;
7 $\boldsymbol{\alpha} \leftarrow G(SK_{i,a})$;
8 $a \leftarrow \boldsymbol{\alpha} \cdot \texttt{truncate}(\boldsymbol{p}_D^f, l + i - 1)$;
9 $b \leftarrow \sum_{j=1}^{m} F(K_c, j) \cdot F(SK_{i,b}, \texttt{VID}||S_{i-1}||j)$;
10 **if** $a + b \neq \boldsymbol{p}_D^f(l + i)$ **then**
11 **return** False;
12 **end**
13 **end**
14 **return** True;

Dealing with packet losses. In the above, we have assumed there are no packet losses, and the destination switch always compresses the same packets as the source switch. However, when some packets are lost, the destination switch will compress less packets than the source switch, resulting in false positives during verification. We approach this problem by dividing a flow into multiple batches, and apply the flow-level REV method separately on each individual batch. When some packets of a batch are lost, only the verification of the same batch is affected. We will analyze the relationship among accuracy of flow-level REV with packet loss rate and batch size in Section IV-B.

C. Network-Level REV

Different from packet-level REV and flow-level REV which only verify the enforcement of a subset of rules in the network, network-level REV aims to verify the enforcement of all rules that are currently matched by some active flows.

A straightforward way is to simply apply flow-level REV on each active flow. However, this is both costly and unnecessary considering the fact that each flow can match multiple rules, and each rule can be matched by multiple flows. In this paper, we aim to achieve network-level REV by selecting a minimum number of flows. Formally, we are going to solve the following problem:

$$\begin{aligned} \min \quad & \sum\nolimits_{f\in\mathcal{F}} x_f \\ \text{s.t.} \quad & \sum\nolimits_{r\in R_f} x_f \geq 1;\ \forall r \in \bigcup_{f\in\mathcal{F}} R_f \\ & x_f = 0,1;\ \forall f \in \mathcal{F} \end{aligned} \tag{6}$$

ATPG [23] studied a similar problem. i.e., finding the minimum number of *test packets* to exercise all rules in a network. The authors showed the problem of finding the minimum number of test packets can be reduced to the classic minimum set cover problem, which is NP. Our problem differs from it in that ATPG can generate arbitrary test packets and theoretically can cover all the rules, REV can only rely on real flows that are active, and thus it may be infeasible to cover all the rules. To achieve network-level REV, we use a heuristic algorithm as shown in Algorithm 2. F is a set that records currently selected flows, and is initialized to be empty (Line 1); U_f is the set of rules that are matched by flow f but not by any flow in F, and it is initialized to R_f, i.e., the set of rules matched by f (Line 2). In each iteration, a flow is selected and F, $\mathcal{F}$, U_f are updated (Line 3-10).

Algorithm 2 `SelectFlows`($\mathcal{F}$)

Input: $\mathcal{F}$: the set of all flows in the network.
Output: F: the set of selected flows for verification.

1 $F \leftarrow \{\}$;
2 $U_f \leftarrow R_f$, for $\forall f \in \mathcal{F}$;
3 **while** $\exists f \in \mathcal{F}$ *s.t.* $|U_f| > 0$ **do**
4 select $f \in \mathcal{F}$, s.t. $|U_f| \geq |U_{f'}|$, for $\forall f' \in \mathcal{F}$;
5 $F \leftarrow F \cup \{f\}$;
6 $\mathcal{F} \leftarrow \mathcal{F} \backslash \{f\}$;
7 **foreach** $f' \in \mathcal{F}$ **do**
8 $U_{f'} \leftarrow U_{f'} \backslash U_f$;
9 **end**
10 **end**

After the flow set F is selected, we can apply the flow-level REV method for each $f \in F$. Another option is to sample some packets of f, and apply the packet-level REV method for each sampled packet. Note that if an adversary can distinguish sampled packets, it can just modify the paths of packets that are not sampled. This is similar to the *coward attack* [39] in VANET. To defend against such coward attack, we require that all packets should carry tags, and each switch should update the tags of all packets. Sampling is done at the source switches, which only send sampled packets to the controller for verification.

IV. Analysis

A. Security

This section analyzes the security of the flow-level REV method. First, an informal theorem regarding the compressive MAC is given as follows:

Theorem 2: Assume that F and G are secure pseudorandom function and generator, respectively. Then, the probability that a probabilistic polynomial-time adversary can forge a tag of packet for a non-compromised switch, such that the verification of Algorithm 1 passes, is negligibly larger than $1/q$, where q is the size of finite field being used.

The above theorem directly follows from Theorem 1 in Appendix (see supplementary file). Based on this theorem, the following shows how the flow-level REV method can defend against the attacks outlined in Section II-B.

Switch Bypass. Suppose a packet p is received by a compromised switch S_i, which forwards p directly to switch S_{i+2}, thereby bypassing the original next hop S_{i+1}. The deviated path is then $S_i \rightarrow S_{i+1} \rightarrow S_{i+2}$, where at least one of S_{i+1} and S_{i+2} is not compromised, according to our assumption. First, suppose S_{i+1} is not compromised, then according to Theorem 2, the probability that an adversary can forge a tag for S_{i+1}, such that the tag passes the verification, is negligibly larger than $1/q$. Second, suppose S_{i+1} is compromised, but S_{i+2} is not compromised. Since a tag encodes the last-hop switch identifier, according to Eq. (5), the probability that an adversary can forge a tag for S_{i+2}, such that the tag passes the verification, is negligibly larger than $1/q$. Thus, the switch bypass can be detected.

Path Detour. Suppose a packet p is received by a compromised switch S_i, which forwards p along a detoured path $D_1 \rightarrow \ldots \rightarrow D_m \rightarrow S_{i+1}$. The deviated path is then $S_i \rightarrow S_{i+1}$, where S_{i+1} is not compromised, according to our assumption. There are two cases. (1) D_m is not compromised. In this case, p will arrive at S_{i+1} with at least one tag generated by D_1 through D_m. Since the tag generated by S_{i+1} will be based on all tags of p according to Eq. (5), it will be different from what would be generated if p is directly received from S_i. Even some downstream switch removes the extra tags generated by D_1 through D_m to make the total number of tags equal to n, the tags generated by S_{i+1} through S_n would still fail the verification with high probability. (2) D_m is compromised. In this case, D_m can remove all the tags generated by D_1 through D_{m-1}, to give the illusion that p is directly forwarded to S_{i+1}. However, since S_{i+1} receives p from a port different from the original one, the tag generated by S_{i+1} will also be different. Thus, path detour can be detected.

Out-of-order Traversal. Suppose a packet p has traversed the original path $S_i \rightarrow \ldots \rightarrow S_{i+k}$, but in a different order. The deviated path is then $S_i \rightarrow \ldots \rightarrow S_{i+k}$, and suppose switch $S_j (i < j \leq i+k)$ is not compromised. Since S_j receives p from another switch other than S_{j-1}, the tag generated by S_j will also be different from the original one. Thus, tags generated under out-of-order traversal would fail the verification with high probability, meaning that out-of-order traversal can be detected.

Unauthorized Access. Suppose a packet p is not authorized to reach an end host, according to some ACL rules. Let the path of p be $S_1 \rightarrow \ldots \rightarrow S_m$, where S_m is the switch dropping the packet. To verify the enforcement of such kind of ACL rules, packet-level REV requires the switch that drops packet p on the forwarding path should send a verification report.

In this sense, the dropping switch takes the role of a destination switch. As for flow-level REV, the controller should send the compression key to the dropping switch, which will send the flow-packet of the dropped flow to the controller when the verification session ends. Here in this example, S_m would report the hash and tag of p before dropping p. If the verification passes for p, then we say that the ACL rules are correctly enforced by p. Note that not all dropping switches should send verification reports. Instead, only those switches which drop flows under verification should send verification reports to the REV server.

Here we implicitly assumed that S_m should drop the packet p. To ensure p is indeed dropped, we can install dedicated rules matching p at the adjacent switches of S_m. For example, if the header of p is h, then we add a rule into each adjacent switch of S_m, with matching fields of $header = h$ and $input\_port = port(S_m)$. If the counters of these rules are all zero, we can ensure p is indeed dropped by S_m.

Packet Replay. Suppose a compromised switch records packets of previous sessions, and replays the packets into the network. Since REV would use `TimeStamp` to derive a different session identifier `VID`, and compute a different session key from `VID`, the tags from previous sessions will not pass the verification. Similarly, replaying a packet within the same session can be prevented by the sequence number in the REV header.

B. Accuracy

In the following, we will analyze the accuracy of REV, and study the tradeoff between accuracy and bandwidth cost.

For the packet-level REV, there are no false positives and the false negative rate is $(1 - 1/q)$, where q is the size of finite field $\mathbb{F}_q$. For the flow-level REV, when there are no packet losses, the accuracy of flow-level REV is the same with that of the packet-level REV. When there are packet losses, the false positive rate is related to the packet loss rate p and batch size b. Specifically, suppose a flow has f packets, then the false positive rate for the nth batch, where $n < f/b + 1$, is calculated as:

$$FPR_n = (1 - 1/q)\big(1 - (1 - p)^{\min(b, f-(n-1)b)}\big) \qquad (7)$$

There is a tradeoff between false positive rate and bandwidth cost. Specifically, the bandwidth cost on the control channel is $\lceil f/b \rceil (nt + 48)$, where n is the number of hops, and t is the length of a tag (see Section VI-G for detailed analysis). Reducing the batch size b will reduce the false positive rate but increase bandwidth cost. For a specific network, a proper value of b depends on the packet loss rates of network links, and the requirement on false positive rate.

V. IMPLEMENTATION

Before introducing the prototype of REV, we first show how the cryptographic operations are implemented, and specify the header format of REV.

Cryptographic operations. We instantiate the pseudorandom number generators G with AES-128 in counter mode, the pseudorandom function F with AES-128 in CBC mode, and the cryptographic hash function H with SHA1.[1] Both AES and SHA1 are implemented using OpenSSL [40]. Computation over Galois field is performed using the fast Galois field arithmetic library [41], and a multiplication table is pre-computed for fast multiplications.

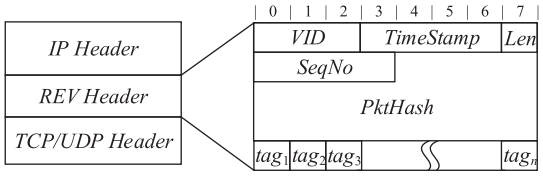


Fig. 3. Header format in the flow-level REV.

REV header format. As shown in Fig. 3, the REV header sits in-between the IP and TCP/UDP header. Since we directly use the output of SHA1 which has 160 bits, the `PktHash` field has a length of 20 Bytes. `Len` stores the number of tags in the current header, and each tag is t Bytes in length (currently we set $t = 1$). `SeqNo` is used by flow-level REV to track the sequence number of packets within the same flow.

Our REV prototype includes two components: *REV server* and *REV datapath*, which consists of *REV agent* and *REV packet processing module*.

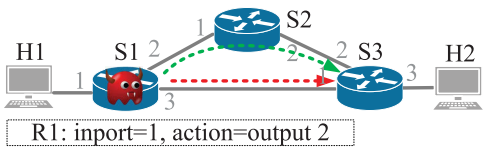
REV Server. We implement the core functions of REV server (e.g., key generation and packet verification) using C, and deploy it as an application atop the SDN controller. Currently, REV supports two controllers: Ryu [16] based on Python, and Floodlight [42] based on Java. We use `ctype` and `JNI` to let the python codes and java codes interface with the C codes, respectively. The REV server application exposes a REST API that allows operators to issue *verification queries*. A verification query specifies the *verification flow* i.e., the flow to be verified, and the *batch size*, i.e., how many packets should be compressed for verification. For example, the following command will check the flow from 10.0.0.1 to 10.0.0.2, with a batch size of 1000:

```
curl -X PUT '{"dst_ip":10.0.0.1, "src_ip":
10.0.0.2, "batch":1000}' 127.0.0.1:8080
/rev/check_flow/
```

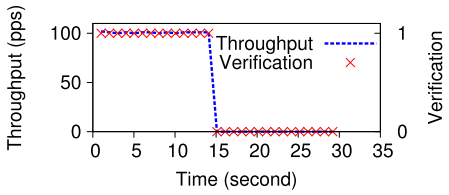
The REV server maintains a *path table* that records all the end-to-end paths, indexed by entry port (source switch ID and local port ID) and packet header (e.g., TCP 5-tuple). The path table can be constructed using the method introduced in [26]. On receiving a verification query, the server looks up in the path table to determine the path of the flow, and sets up a *verification session*. By doing so, the server sends the information of the verification flow and the batch size to the first and last switch on the path. OpenFlow `experimenter` message is used for communication among the REV server and switches.

REV Datapath. We implement the REV datapath using OVS-DPDK, i.e., Open vSwitch with DPDK data path [30]. Two extra components are added: (1) a REV agent which communicates to the server with `experimenter` messages,

[1]REV does not rely on specific choice of hash functions, and other hash functions like SHA256 and SHA512 can also be used.



(a) The testbed for rule modification attack.



(b) Packet throughput at $S2$ and verification results. "0" and "1" means verification fails and passes, respectively.

Fig. 4. Detection of rule modification attack using REV.

in order to set up symmetric keys with the server, initiate verification sessions, and send compressed packets/tags; (2) a REV packet processing module, which is responsible for inserting, removing, parsing REV headers, and computing, compressing, sending MACs. The source switch maintains a table of all active verification sessions. If a packet belongs to a verification session, the source switch inserts a REV header into the packet. All downstream switches just inspect whether there is a REV header in order to determine whether to generate a MAC for the packet. When the sequence number reaches the batch size, or the session times out, both the source and destination switches would send their respective flow-packets to the REV server.

VI. Evaluation

This section first presents a functional test of REV, and then evaluates REV in terms of verification performance, data plane performance, flow selection efficiency, bandwidth overhead, computational overhead, and accuracy.

A. Dataset

In some of the experiments, we will use a sample trace of a backbone router, collected in 2011 [43]. The trace has a volume of 28,475MB, consisting of 935,365 flows and 37,571,701 packets. The average packet size is 757 Bytes, and the average flow size is 40, i.e., each flow consists of 40 packets on average. Later, when we refer to the *trace data*, we mean the specific configuration of $f = 40$ and $p = 757$ Bytes.

B. Functional Test

In this experiment, we show how REV can detect rule modification attack using a real attack scenario. We set up a physical testbed consisting of three switches and two hosts, as shown in Fig. 4(a). Each switch is realized with Open vSwitch running the DPDK data path, and each port of switch is bound to a physical NIC. We let host $H1$ inject traffic to $S1$ at a rate of 100 packets per second. We pre-install rules at these switches so that the packets will be forwarded along the path $S1 \rightarrow S2 \rightarrow S3$. We check the enforcement of these rules at the REV server, and at the same time monitor the packet throughput at $S2$. After 15 seconds, we modify the rule $R1$, so that $S1$ forwards the subsequent packets directly to $S3$, bypassing $S2$. Fig. 4(b) shows that initially the packet throughput at $S2$ stays at around 100 packets per second, while drops to 0 after 15 seconds, meaning the attack indeed works. In addition, the verification passes for all packets in the beginning, while fails after 15 seconds, meaning that REV can successfully detect the attack.

C. Verification Performance

This experiment evaluates the verification performance in terms of throughput and delay. To conveniently vary the number of hops, we use a linear topology consisting of one source switch, n core switches, and one destination switch. We generate a flow of f packets, each of which has p Bytes, and feed all these packets to the source switch. Then, the destination switch outputs a compressed hash and n compressed tags. Finally, we let the REV server verify the compressed hash and tags, and record the time t. The experiment runs on a Linux server with a 3.6GHz Intel I7 processor and 32GB memory. The throughput is defined as the total number of packets f divided by the verification time t, and the delay is defined as the time to verify a single packet hash and tag.

Fig. 5(a) reports the verification throughput for different number of hops. It shows that the throughput drops when the flow path consists of more hops. The reason is that each core switch would append a tag to each packet, and the server needs to verify each of the tags. The throughput is around 1Mpps for the trace data when there are 8 hops.

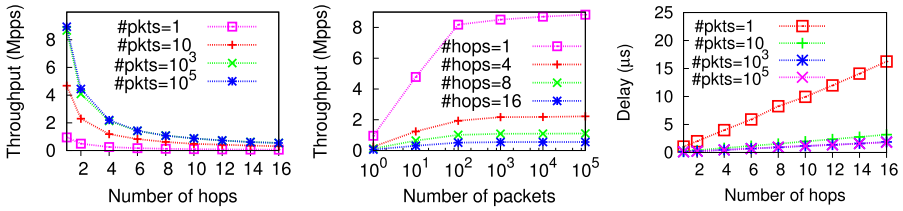
Fig. 5(b) reports the relationship between verification throughput and flow size. We can see that the throughput is rather low (0.12Mpps) when there is only one packet in the flow. This corresponds to the packet-level REV method, where each packet needs to be verified individually. On the other hand, the throughput increases to around 0.92Mpps when there are 8 hops. This indicates that with compressive MAC, the flow-level REV method can achieve a nearly 8× increase in verification throughput.

Fig. 5(c) reports the verification delay at the REV server. We can see that the delay of the REV server is impacted by flow size, and is rather small when the flow has more than 10 packets: it takes around 3.2μs to verify a packet when there are 16 hops.

D. Datapath Performance: Micro-Benchmark Results

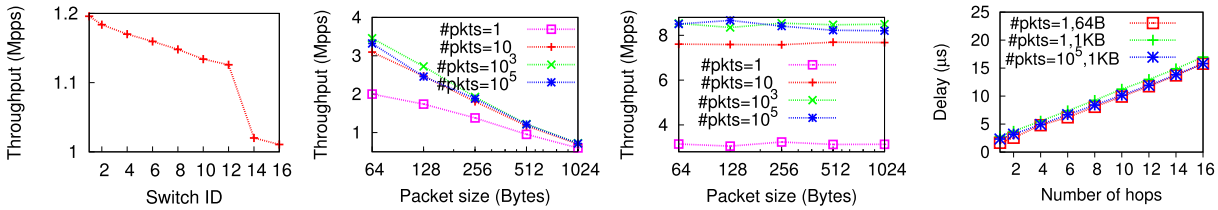
In this experiment, we micro-benchmark the datapath performance by extracting the computation logics of REV as a stand-alone program, and measuring the processing throughput and delay. For throughput, we distinguish among three different types of switches: source switches, destination switches, and core switches. We continue to use the experiment setup of Section VI-C.

Fig. 6(a) reports the throughput of core switches, when the flow consists of 16 hops (i.e., core switches). Since the computation cost of a core switch depends on the number of



(a) verification throughput (b) verification throughput (c) verification delay

Fig. 5. Verification throughput and delay.



(a) switch throughput (core) (b) switch throughput (source) (c) switch throughput (dest.) (d) delay

Fig. 6. The micro-benchmark for datapath throughput and delay of REV.

tags carried by a packet, the throughput decreases as the switch identifier increases (switches are numbered according to their appearance on the path). The reason for the steep drop between 12 and 14 hops is as follows. First, recall that the ith core switch along the path uses pseudorandom number generator (PRNG) to derive a vector α_i of length $(20 + i)$ Bytes (see Section III-B), where 20 is the length of packet hash. Since we use AES-128 (block size of 16 Bytes) in counter mode to implement the PRNG, two AES blocks are encrypted when $i \leq 12$, and three blocks are encrypted when $12 < i \leq 28$. Therefore, the computational overhead increases and packet throughput drops sharply when i increases from 12 to 14 in Fig. 6(a). Note the throughput has no relationship with either packet sizes or flow sizes, thus we do not report them here. We can see that the processing throughput is around 1Mpps when there are no more than 16 hops. This translates into a data throughput of $757 \times 8 \times 10^6 = 6$Gbps for the trace data.

Fig. 6(b) and Fig. 6(c) report the throughput of source and destination switch, respectively. We can see that the throughput of the source switch is around 0.95Mpps for the trace data (packet size = 757 Bytes). Since the source switch needs to compute packet hashes, its throughput drops when the packet size grows. On the other hand, the throughput of the destination switch is much higher and does not change with packet size, as it does not need to compute packet hashes. In particular, the throughput is around 7.6Mpps for the trace data, 8 times that of the source switch.

Fig. 6(d) reports the delay of the REV datapath. The delay represents the sum of processing time at all switches, which is also very small and barely related to packet size and flow size. It takes around 16μs when there are 16 hops. The results show that the flow-level REV incurs minimal latency on the data path.

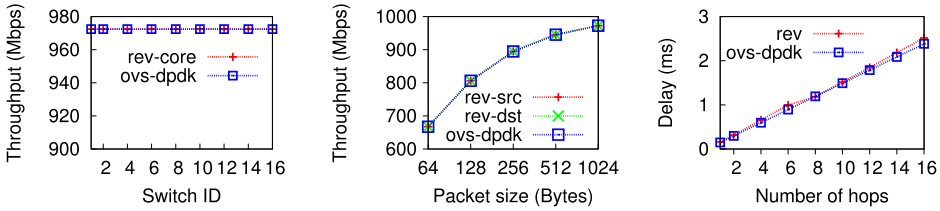
E. Datapath Performance: Testbed-Based Results

The above micro-benchmark results only validate the computation efficiency of REV when running alone on dedicated servers, while may not ensure it can process packets efficiently when implemented in real switches. Thus, we complement the micro-benchmark results by evaluating the performance of REV datapath implemented based on OVS-DPDK. Specifically, we set up a small physical testbed consisting of three servers. The first server H hosts an OVS-DPDK switch running the REV datapath, the second server generates traffic to one port of H, and the third server $S3$ receives the traffic from another port of H. We use `netmap`, a fast packet I/O [44], to measure the throughput. All the three servers have the same configuration: two 2.0GHz Intel Xeon-E5 CPUs, 32GB memory, and a dual-port 1Gbps NIC.

Fig. 7(a) itches for different hops, when the packet size is 1024 Bytes. To simulate the nth hops, we let Server 2 add $(n-1)$ tags to each packet, and run the tagging process on it. We can see that both REV and plain OVS-DPDK can achieve a throughput of around 970Mbps when there are less than 16 hops. Fig. 7(b) shows the throughput of edge switches for different packet sizes. We can see that the throughput of edge switches is roughly the same with plain OVS-DPDK, increasing to around 970Mbps when the packet size reaches 1024 Bytes. The above results means that REV can achieve near line rate processing on 1Gbps NICs. Finally, Fig. 7(c) shows that the delays of plain OVS-DPDK and OVS-DPDK with REV are roughly the same, both around 2.5ms when there are 16 hops. This means REV adds minimal overhead to the data path processing delay, therefore has little impact on the QoS of flows.

F. Flow Selection Efficiency

We evaluate how Algorithm 2 can efficiently select active flows to cover as many rules as possible. Here, we use the Stanford backbone network topology [45], which consists of 16 routers and 10 Layer-2 switches, and two datacenter topologies, i.e., FatTree(4) and DCell(1,4). For each topology, we let each host ping one another to populate the switches' flow tables. Then, for each switch we aggregate those rules



(a) throughput (core) (b) throughput (edge) (c) delay

Fig. 7. The testbed evaluation for datapath throughput and delay of REV.

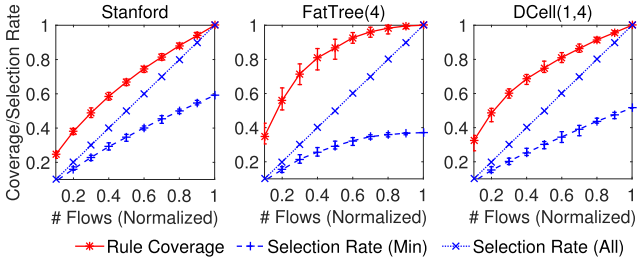


Fig. 8. The selection rate and coverage rate of the network-level REV for three different network topologies. The x-axis is the number of flows, normalized as the ratio of active flows to all the flows; the y-axis is either the coverage rate, defined as the ratio of the covered rules to all the rules, or the selection rate, defined as the ratio of the selected flows to all the active flows. "All" refers to selecting all active flows, and "Min" refers to selecting flows using our heuristic algorithm.

with the same IP destination address. During experiments, we vary the number of active flows, by randomly selecting a different number of pairs of hosts, and running `iperf` between them. Thus, there are $N = n(n-1)$ possible flows in total.

Fig. 8 reports the *coverage rate*, defined as the ratio of number of covered rules to number of all rules, and the *selection rate*, defined as the ratio of number of selected flows to number of all active flows, for different number of active flows. Here, "All" refers to selecting all active flows, and "Min" refers to select flows using our heuristic algorithm. We can see that for both selection strategies, the coverage rate increases to 1 as there are more active flows. More importantly, by using the heuristic flow selection algorithm (Algorithm 2), we can reduce the number of selected flows by 40%-60% when all flows are active.

G. Bandwidth Overhead

In the following, we evaluate the per-flow bandwidth overhead of flow-level REV which uses compressive MAC, and compare it with those of packet-level REV and SDNsec [28], which both use standard MAC. Without loss of generality, we consider a flow which traverses n hops (excluding the source and destination switch) and consists of f packets in total.

First let us consider the flow-level REV method. In the data plane, each packet carries a fixed-length header of 32 Bytes, and a variable number of n tags. As each tag has t Bytes ($t = 1$ in current implementation), the bandwidth overhead per flow is $(nt + 32)f$ Bytes. For the control channel, both

TABLE I

PER-FLOW BANDWIDTH OVERHEAD FOR FLOW-LEVEL REV WHICH USES COMPRESSIVE MAC (REV-FLOW), PACKET-LEVEL REV WHICH USES STANDARD MAC (REV-PACKET), AND SDNSEC [28] WITH ONLY THE PATH VALIDATION COMPONENT (SDNSEC-PVC). t IS THE TAG SIZE IN BYTES, f IS THE NUMBER OF PACKETS OF A FLOW, AND n IS THE NUMBER OF SWITCHES ON THE FORWARDING PATH (EXCLUDING THE SOURCE AND DESTINATION SWITCHES). FOR SDNSEC, WE ONLY COUNT THE FIELDS USED FOR PVC

	SDNsec-PVC	REV-Packet	REV-Flow
Control Channel	$(t+6)f$	$(t+48)f$	$nt+48$
Data Plane	$(t+6)f$	$(t+32)f$	$(nt+32)f$

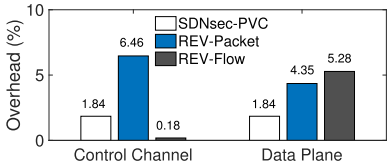


Fig. 9. Bandwidth overhead for SDNsec-PVC, packet-level REV, and flow-level REV, respectively, when the number of hops is set to 8. The results are based on a backbone router trace from CAIDA [43].

the source switch and destination switch should send `VID` and the compressed hash, which have 24 Bytes. In addition, the destination switch should send `Len` and n compressed tags. Thus, the total bandwidth overhead per flow is $24+(nt+24) = (nt + 48)$ Bytes.

Then, let us consider the packet-level REV method. In the data plane, since only one tag is needed for each packet, the bandwidth overhead per flow is $(t + 32)f$ Bytes. For the control channel, both the source and destination switch should send a verification report for each packet of the flow. Then, the bandwidth overhead per flow is $(t + 48)f$ Bytes.

Finally, let us consider SDNsec [28]. For fair comparison, we only consider the path validation component (PVC), which requires three fields, i.e., the path validation field (PVF) of $t = 8$ Bytes, the flow ID, and sequence number, both of which have 3 Bytes. Similar to packet-level REV, SDNsec has a bandwidth overhead of $(t + 6)f$ Bytes in the data plane. For the control channel, since only the destination switch needs to report PVF, flow ID, and sequence number, the overhead is then $(t + 6)f$. The above results are summarized in Table I.

Fig. 9 shows the bandwidth overhead for flow-level REV, packet-level REV, and SDNsec with path validation

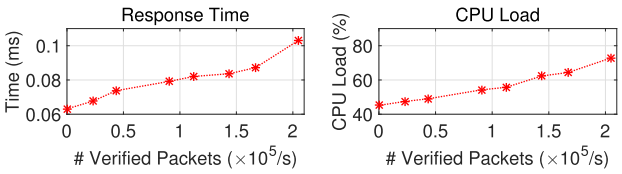


Fig. 10. The computational overhead imposed by REV on SDN controller.

component only. We continue to use the trace data described in Section VI-A, and set the number of hops to 8. We can see that the flow-level REV reduces the bandwidth cost of the control channel by around 97%, compared with packet-level REV, and by around 90% compared with SDNsec. For the data plane, the bandwidth overhead of the flow-level REV is slightly higher than the other two methods due to the usage of multiple tags. However, the overhead is still quite low (5.28%).

H. Computational Overhead

Since REV runs as an application atop SDN controller, it introduces some computational overhead on the controller. In the following, we evaluate such overhead with experiments. Specifically, we run the REV server atop the Floodlight controller, which is hosted on a Linux server with a 3.3GHz Intel Xeon processor (a single core is used). To simulate normal controller processing, we run Mininet on another server to generate `PacketIn` messages to the controller, at a speed of 3000/s. On receiving a `PacketIn` message, the controller calculates a path and installs a set of rules to the corresponding switches. We measure the response time for `PacketIn` messages as the elapsed time from when a `PacketIn` message is received to when a set of rules are sent by the controller. We also monitor the server CPU load during the process. The number of hops is set to 16, and the number of packets per flow is set to 10,000. The experiment lasts for 120 seconds.

Fig. 10 reports the controller's average response time for `PacketIn` messages, and the average CPU load of the server running the controller. We can see that both the response time and CPU load grow slowly, with the response time being less than 0.12ms, and CPU load being lower than 80%, when the verification load is as large as 2×10^5 packets per second. This demonstrates that REV incurs a small overhead on controller under a moderate verification load. In addition, the small increase in controller's response time has little impact on the quality of service (QoS) of normal flows, considering the round trip time of flows is mostly at a scale of several milliseconds.

Finally, Table II compares the computational overhead of REV with SDNsec-PVC. We can see that the flow-level REV uses slightly more cryptographic operations than SDNsec-PVC: the source switch of REV needs to compute the hash for a packet while SDNsec-PVC does not compute hashes; the core switch of REV encrypts four 16-Byte blocks with AES, while SDNsec-PVC only encrypts one block.

I. Accuracy

As noted in Section IV-B, there is a tradeoff between false positive rate (FPR) and bandwidth cost for the flow-level REV.

TABLE II
PER-PACKET COMPUTATIONAL OVERHEAD FOR FLOW-LEVEL REV AND SDNSEC [28] WITH ONLY THE PATH VALIDATION COMPONENT (SDNSEC-PVC)

Operation	SDNsec-PVC	REV(src)	REV(core)	REV(dst)
AES	1	1	4	1
SHA	0	1	0	0

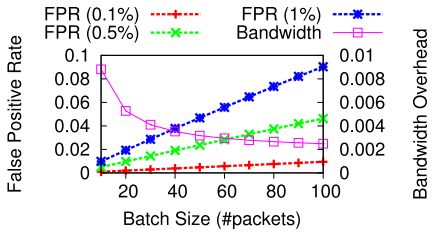


Fig. 11. The relationship between false positive rate and bandwidth overhead for the flow-level REV.

In this experiment, we will study such tradeoff using the trace data. We choose three different packet loss rates (0.1%, 0.5%, and 1%), and vary the batch size from 10 to 100. Fig. 11 reports the average FPR computed with Eq. (7), and bandwidth overhead on the control plane, for different batch sizes. We can see that FRP increases roughly linearly with batch size. On the other hand, the bandwidth overhead quickly drops from 0.88% to 0.41% when the batch size increases from 10 to 30, and begins to drops slowly afterwards. The false positive rate is around 0.3% for packet loss rate of 0.1% and batch size of 30. This suggests that a small batch size like 30 is a good choice for this trace when the packet loss rate is small.

VII. RELATED WORK

Vulnerability of SDN data plane. Pickett [7] finds that the operating systems running on white-box SDN switches are vulnerable to attacks. Moreover, Pickett shows that even the boot loader of the operating systems can be compromised, allowing an attacker gain persistent control over an SDN switch. Antikainen et al. [8] show how to conduct a series of attacks in SDN, by modifying rules of SDN switches. Chi et al. [9] introduce some attacks that can be launched by malicious SDN switches, including incorrect forwarding, duplicated forwarding, packet manipulating, and malicious weight adjusting. Many studies try to detect malicious SDN switches by checking the consistency of flow statistics [10], [12], [46]–[48]. However, these approaches can only achieve a coarse granularity in verification, but cannot ensure that each packet has been correctly forwarded.

Data plane verification and testing. Many tools have been proposed recently to verify the correctness of data plane configuration [19]–[22]. However, even the data plane configuration is correct, switches may still experience faults. ATPG [23] addresses this issue by actively testing the data plane with probe packets. However, ATPG only checks pairwise reachability, thereby cannot detect faults that do not hurt reachability while changing the forwarding paths.

VeriDP [26], [27] addresses this issue by checking whether the packet forwarding behaviors are complying with the data plane configuration. However, both ATPG and VeriDP assume switches are trustable, and thus cannot be used for rule enforcement verification. Monocle [24], RuleScope [25], and RuleChecker [49] check whether rules in a switch' flow table are consistent with those expected by the controller, by injecting probe packets. Similar to ATPG and VeriDP, they also assume switches are trustable, and thus cannot work under adversarial settings.

Path verification. Path verification for the Internet has been extensively studied [35], [50], [51]. The basic idea is to let each router along the forwarding path embeds a cryptographic tag, e.g., Message Authentication Code (MAC), into each packet, such that destination switch can check whether a packet has followed the path claimed by the sender. SDNsec [28] checks the path enforcement in SDN data plane. However, to check whether every packet of flow has followed the correct path, the egress switch needs to report the PVC header for each packet to the controller. This can result in a large bandwidth overhead on the control channel. In contrast, REV uses compressive MACs to greatly reduce the bandwidth overhead. Different from SDNsec that checks path enforcement of packets, WedgeTail [11] aims to detect malicious forwarding node. The problem with WedgeTail is that it may not be able to detect a malicious switch with only few modified rules, which can be detected by both REV and SDNsec.

Message Authentication Codes. There are many variants of Message authentication code (MAC), of which aggregate MAC [52] and homomorphic MAC [36] are most relevant to our compressive MAC. Aggregate MACs [52] are designed for scenarios where one receiver shares different keys with multiple senders. Using aggregate MACs, tags of multiple senders for their respective packets can be aggregated into a single tag, such that the receiver can verify the integrity of all the packets with this tag. Comparatively, the compressive MAC allows not only tags, but also packets, to be compressed. In addition, the compressive MAC only considers the scenario where there is only one sender, and both the sender and receiver should report packets and tags to the verifier. Homomorphic MACs [36] allow switches to linearly combine packets and compute tags for the combined packets, without knowing the MAC key shared between the sender and receiver. Compressive MAC shares the same spirit with homomorphic MACs in that they both allow packets and tags to be combined. However, homomorphic MACs can only guarantee the received packets are linear combinations of source packets, while compressive MAC ensures that the received packets have traversed the intended path.

VIII. Discussion

Security beyond $1/q$. In the current implementation of REV, each MAC has a single byte, i.e., an element of finite field $\mathbb{F}_{2^8}$. However, our construction of compressive MAC has no assumption of finite field size q. Thus, if $q = 2^{8k}$, each MAC will have k Bytes. Then, the probability of forging a tag is then $1/2^{8k}$, which can be made arbitrarily small by choosing a sufficiently large k. We can also associate each hop with multiple ($k > 1$) single-byte tags. Then, each hop can be seen as being expanded into k hops, thereby generating k tags for each packet. This will also reduce the probability of tag forging to $1/2^{8k}$. The proof easily follows from the proof for a single tag in the Appendix (see supplementary file).

IX. Conclusion

This paper proposed rule enforcement verification (REV), a new security primitive that can defend SDNs against rule modification attacks. Using REV, the controller can verify whether rules had been correctly enforced by switches. To reduce the switch-to-controller traffic of each flow, we proposed compressive MAC, a new MAC that allowed tag compression. To efficiently verify all rules in the network, we proposed the network-level REV, which used a heuristic algorithm to select a small number of flows for verification. Experiments showed: (1) by using compressive MAC, REV can achieve a 97% reduction in switch-to-controller traffic and an $8\times$ increase in verification throughput, and (2) the network-level REV can reduce the number of flows for verification by 40%-60%.

References

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

[2] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[3] (2005). *Cisco Security Hole a Whopper*. [Online]. Available: https://www.wired.com/2005/07/cisco-security-hole-a-whopper/

[4] (2014). *Snowden: The NSA Planted Backdoors in Cisco Products*. [Online]. Available: http://bit.ly/1PKtbQW

[5] (2015). *Cisco Routers Compromised by Malicious Code Injection*. [Online]. Available: http://bit.ly/1KtUoTs

[6] *FBI: Reboot Your Home and Small Office Routers to Counter Russian Malware*. Accessed: May 27, 2018. [Online]. Available: https://www.csoonline.com/article/3276270/fbi-reboot-your-home-and-small-office-routers-to-counter-russian-malware.html

[7] G. Pickett, "Staying persistent in software defined networks," in *Proc. Black Hat Briefings*, 2015, pp. 1–9. Accessed: Dec. 2018. [Online]. Available: https://www.blackhat.com/docs/us-15/materials/us-15-Pickett-Staying-Persistent-In-Software-Defined-Networks-wp.pdf

[8] M. Antikainen, T. Aura, and M. Särelä, "Spook in your network: Attacking an sdn with a compromised openflow switch," in *Proc. Nordic Conf. Secure IT Syst.*, 2014, pp. 229–244.

[9] P.-W. Chi, C.-T. Kuo, J.-W. Guo, and C.-L. Lei, "How to detect a compromised SDN switch," in *Proc. 1st IEEE Conf. Netw. Softwarization (NetSoft)*, Apr. 2015, pp. 1–6.

[10] T.-W. Chao *et al.*, "Securing data planes in software-defined networks," in *Proc. IEEE NetSoft Conf. Workshops (NetSoft)*, Jun. 2016, pp. 465–470.

[11] 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. (ASIA CCS)*, 2017, pp. 849–861.

[12] P. Zhang *et al.*, "FOCES: Detecting forwarding anomalies in software defined networks," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2018, pp. 830–840.

[13] *Openflow Switch Specification (Version 1.5.1)*, Open Netw. Found., Palo Alto, CA, USA, 2015.

[14] K. Benton, L. J. Camp, and C. Small, "OpenFlow vulnerability assessment," in *Proc. 2nd ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2013, pp. 151–152. [Online]. Available: http://homes.soic.indiana.edu/ktbenton/research/openflow_vulnerability_assessment.pdf.

[15] *The ONOS SDN Controller*. Accessed: Mar. 2020. [Online]. Available: https://www.opennetworking.org/onos/
[16] *The Ryu OpenFlow Controller*. Accessed: Mar. 2020. [Online]. Available: https://osrg.github.io/ryu/
[17] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: Validating SSL certificates in non-browser software," in *Proc. ACM Conf. Comput. Commun. Secur. (CCS)*, 2012, pp. 38–49.
[18] *The Heartbleed Bug*. Accessed: Mar. 2020. [Online]. Available: https://heartbleed.com/
[19] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proc. NSDI*, 2013, pp. 15–27.
[20] H. Yang and S. S. Lam, "Scalable verification of networks with packet transformers using atomic predicates," *IEEE/ACM Trans. Netw.*, vol. 25, no. 5, pp. 2900–2915, Oct. 2017.
[21] A. Horn, A. Kheradmand, and M. R. Prasad, "Delta-net: Real-time network verification using atoms," in *Proc. USENIX NSDI*, 2017, pp. 735–749.
[22] K. Jayaraman *et al.*, "Validating datacenters at scale," in *Proc. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2019, pp. 200–213.
[23] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," *IEEE/ACM Trans. Netw.*, vol. 22, no. 2, pp. 554–566, Apr. 2014.
[24] P. Pereší ni, M. Kuźniar, and D. Kostić, "Monocle: Dynamic, fine-grained data plane monitoring," in *Proc. 11th ACM Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, 2015, pp. 1–13.
[25] 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. 35th Annu. IEEE Int. Conf. Comput. Commun. (INFOCOM)*, Apr. 2016, pp. 1–9.
[26] P. Zhang, H. Li, C. Hu, L. Hu, and L. Xiong, "Stick to the script: Monitoring the policy compliance of SDN data plane," in *Proc. Symp. Archit. Netw. Commun. Syst. (ANCS)*, 2016, pp. 81–86.
[27] P. Zhang *et al.*, "Mind the gap: Monitoring the control-data plane consistency," in *Proc. ACM CoNEXT*, 2016, pp. 19–33.
[28] T. Sasaki, C. Pappas, T. Lee, T. Hoefler, and A. Perrig, "SDNsec: Forwarding accountability for the SDN data plane," in *Proc. 25th Int. Conf. Comput. Commun. Netw. (ICCCN)*, Aug. 2016.
[29] *CPqD: The OpenFlow 1.3 Compatible User-Space Software Switch*. Accessed: Dec. 2018. [Online]. Available: http://cpqd.github.io/ofsoftswitch13
[30] *Open vSwitch With DPDK Overview*. Accessed: Dec. 2018. [Online]. Available: https://software.intel.com/en-us/articles/open-vswitch-with-dpdk-overview
[31] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "AVANT-GUARD: Scalable and vigilant switch flow management in software-defined networks," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2013, pp. 413–424.
[32] 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. 8–11.
[33] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Trans. Inf. Theory*, vol. IT-22, no. 6, pp. 644–654, Nov. 1976.
[34] E. Barker, L. C. Nist, A. Roginsky, A. Vassilev, and R. Davis. (2018). *Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography*. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-56a/rev-3/final
[35] T. H.-J. Kim, C. Basescu, L. Jia, S. B. Lee, Y.-C. Hu, and A. Perrig, "Lightweight source authentication and path validation," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 271–282, Aug. 2014.
[36] S. Agrawal and D. Boneh, "Homomorphic MACs: MAC-based integrity for network coding," in *Proc. ACNS*, 2009, pp. 292–305.
[37] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," in *Proc. ACM Symp. Theory Comput.*, 1977, pp. 106–112.
[38] P. Zhang, Y. Jiang, C. Lin, H. Yao, A. Wasef, and X. Shenz, "Padding for orthogonality: Efficient subspace authentication for network coding," in *Proc. IEEE INFOCOM*, Apr. 2011, pp. 1026–1034.
[39] B. Liu, J. T. Chiang, J. J. Haas, and Y.-C. Hu, "Coward attacks in vehicular networks," *ACM SIGMOBILE Mobile Comput. Commun. Rev.*, vol. 14, no. 3, p. 34, Dec. 2010.
[40] *The OpenSSL Project*. Accessed: Dec. 2018. [Online]. Available: http://www.openssl.org/
[41] *Fast Galois Field Arithmetic Library in C/C++*. Accessed: Dec. 2018. [Online]. Available: http://web.eecs.utk.edu/~plank/plank/papers/CS-07-593
[42] *Floodlight OpenFlow Controller*. Accessed: Mar. 2020. [Online]. Available: http://www.projectfloodlight.org/floodlight/
[43] *Packet Traces From Wide Backbone*. Accessed: Dec. 2018. [Online]. Available: http://mawi.wide.ad.jp/mawi/samplepoint-F/2011/201101231400.html
[44] L. Rizzo, "Netmap: A novel framework for fast packet I/O," in *Proc. USENIX Secur. Symp.*, 2012, pp. 101–112.
[45] *Hassel, the Header Space Library*. Accessed: Dec. 2018. [Online]. Available: https://bitbucket.org/peymank/hassel-public
[46] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "SPHINX: Detecting security attacks in software-defined networks," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 8–11.
[47] A. Kamisinski and C. Fung, "FlowMon: Detecting malicious switches in software-defined networks," in *Proc. ACM CCS Workshop Automat. Decis. Making Act. Cyber Defense*, 2015, pp. 39–45.
[48] C. Pang, Y. Jiang, and Q. Li, "FADE: Detecting forwarding anomaly in software-defined networks," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2016, pp. 1–6.
[49] P. Zhang, C. Zhang, and C. Hu, "Fast testing network data plane with RuleChecker," in *Proc. IEEE 25th Int. Conf. Netw. Protocols (ICNP)*, Oct. 2017, pp. 1–10.
[50] X. Liu, A. Li, X. Yang, and D. Wetherall, "Passport: Secure and adoptable source authentication," in *Proc. USENIX NSDI*, 2008, pp. 1–14.
[51] J. Naous, M. Walfish, A. Nicolosi, D. Mazières, M. Miller, and A. Seehra, "Verifying and enforcing network paths with ICING," in *Proc. 7th Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, 2011, pp. 1–12.
[52] J. Katz and A. Y. Lindell, "Aggregate message authentication codes," in *Topics in Cryptology—CT-RSA*. Berlin, Germany: Springer, 2008.

Peng Zhang (Member, IEEE) received the Ph.D. degree in computer science from Tsinghua University in 2013. He is currently an Associate Professor with the School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China. His research interests include verification, measurement, as well as privacy and security in computer networks.

Hui Wu received the B.S. degree in computer science from Xi'an Jiaotong University, China, in 2018, where she is currently pursuing the master's degree with the School of Computer Science and Technology. Her research topic is software-defined networking.

Dan Zhang received the M.S. degree in computer science from Xi'an Jiaotong University in 2019. Her research topic is software-defined networking.

Qi Li (Senior Member, IEEE) received the Ph.D. degree from Tsinghua University. He is currently an Associate Professor with the Institute for Network Sciences and Cyberspace, Tsinghua University. He has worked with ETH Zurich, The University of Texas at San Antonio, The Chinese University of Hong Kong, and Chinese Academy of Sciences. His research interests include network and system security, particularly in Internet and cloud security, mobile security, and big data security. He is currently an Editorial Board Member of the IEEE TDSC and ACM DTRAP.