Towards Rule Enforcement Verification for Software Defined Networks

Peng Zhang
Department of Computer Science and Technology
Xi'an Jiaotong University

***Abstract*—Software defined networks (SDNs) reshape the ossified network architectures, by introducing centralized and programmable network control. Despite the huge benefits, SDNs also open doors to what we call *rule modification attack*, an attack largely overlooked by the community. In such an attack, the adversary can modify rules by exploiting implementation vulnerabilities of switch OSes and control channels. As a result, packets may deviate from their original paths, thereby violating network policies. To defend against rule modification attack, this paper introduces a new security primitive named *rule enforcement verification (REV)*. REV allows a controller to check whether switches have enforced the rules installed by it, using message authentication code (MAC). Since using standard MACs will incur heavy switch-to-controller traffic, this paper proposes a new *compressive MAC*, which allows switches to compress MACs before reporting to the controller. Experiments show that REV based on compressive MAC can achieve a 97% reduction in switch-to-controller traffic, and a $8\times$ increase in verification throughput.**

***Index Terms*—Software defined networks, Rule modification attack, verification, Compressive MAC**

I. Introduction

Software defined networks (SDNs) promise a centralized, flexible, and programmable control of computer networks. In a typical SDN, a logically-centralized controller compiles network policies (*e.g.*, routing, access control, waypoint traversal) into forwarding rules, and installs them at switches through a standard control channel (*e.g.*, OpenFlow [1]). Switches enforce these forwarding rules to realize the network policies.

Even SDNs offer many benefits that are absent in traditional networks, they also raise new security issues. First, the operating systems (OSes) of SDN switches are vulnerable to attacks [2]–[4]. For example, a recent study shows that an attacker can compromise the boot loader of switch OSes, so as to persistently control SDN switches [2]. Secondly, the control channels between the controller and switches also lack security protection. Even the OpenFlow standard recommends the usage of SSL/TLS, most SDN switch vendors just forgo this feature [5].

The above security vulnerabilities open doors to what we call *rule modification attack*, a threat vector that is largely overlooked by the SDN community. In rule modification attack, the adversary can either compromise the switch OS, or act as a man-in-the-middle on the control channel, in order to tamper with rules installed by the controller. As a result, packets may deviate from their original paths, violating original network policies. For example, a rule may require a specific flow go through a firewall, and if the attacker can delete this rule, all packets belonging to this flow will bypass the firewall.

Rule modification attack is hard to detect due to the open-looped control model of SDN: controllers only *install* rules at switches, but cannot verify whether these rules are *enforced* by switches. Even there are many verification tools for SDN, most of them aim to check the correctness of network policies at the controller side [6]–[8], or assume switches are trustable [9]–[11]. As a result, they cannot work under the adversarial setting where switches can be compromised.

To this end, this paper motivates *Rule Enforcement Verification (REV)*, a new security primitive for SDNs. At a high level, an REV mechanism should allow a controller to *securely* verify whether rules installed by it have been correctly enforced by switches, thereby dismissing the rule modification attack.

This paper realizes REV based on message authentication code (MAC). In a nutshell, when a packet enters the network, the entry switch reports the packet to the controller, and attaches a tag to the packet. Each downstream switch updates the tag with the secret key shared with the controller. When the tagged packet is about to leave the network, 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. To handle this challenge, this paper proposes *Compressive MAC*, a novel MAC that allows edges 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 they 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. This paper proves that the compressive MAC is secure under standard cryptographic models.

In sum, our contribution is three-fold:

- We motivate rule enforcement verification (REV), which can be used to defend against the rule modification attack in SDN. To the best of the author's knowledge, this is the first study on rule verification problem of SDN under adversarial settings.

- We introduce a new message authentication code, named compressive MAC, and theoretically prove its security under standard cryptographic models.
- We realize REV based on the compressive MAC, and show that it can achieve a 97% reduction in switch-to-controller traffic, and an $8\times$ improvement in verification throughput, compared with using standard MACs.

The rest of this paper proceeds as follows. Section II states the problem of REV. Section III introduces the REV realization. Section IV analyzes the security of the REV realization, and Section VI evaluates its performance. Section VII surveys related work, and Section VIII concludes.

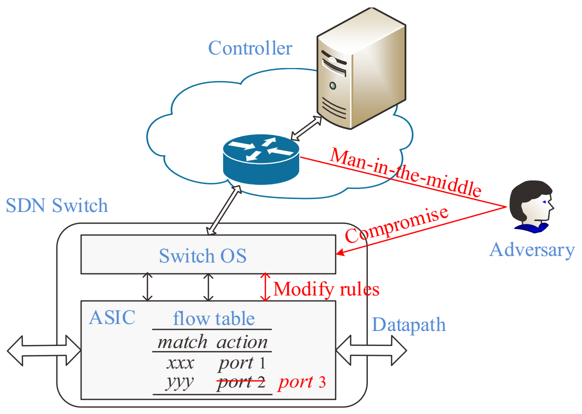


Fig. 1. An illustration of rule modification attack in SDN.

II. Problem Statement

A. Network Model

This paper considers a typical SDN consisting of one *controller* and multiple *switches*. The switches and their links collectively form the SDN *datapath*. 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 the flow table. 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 assumes the adversary can launch *rule modification attack*, as illustrated in Fig. 1. There are two ways for such an attack. First, the adversary can exploit software vulnerabilities of the switch OS, and install backdoor applications on the switch OS. Then, it can install, delete, or modify rules in the hardware flow tables. As the second way, since the connections between the controller and switches may traverse multiple hops, the adversary can reside at intermediate switches as a man-in-the-middle. Then, it can inject, drop, or modify rule installation messages (*e.g.*, flow-mod messages as in OpenFlow) sent by the controller. By launching rule modification attack, the adversary aims at the following two goals:

Path Deviation. Packets take different paths than what are expected 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 attack can break isolation policies, *e.g.*, only users from CS department can access a web server.

This paper assumes that the controller is always trusted, while the switches can be compromised by the adversary. In addition, multiple compromised switches can collude to evade the verification. It is also assumed that the adversary knows the verification methods being used, and can record packets and replay them into the network.

C. Rule Enforcement Verification

Let f be a flow and 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 $Path(\mathcal{R}, f)$.

III. REV Methods

This section presents two concrete methods to achieve REV, one for packet-level REV, and one for flow-level REV.

A. Packet-Level REV

The packet-level REV method consists of 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). The controller shares a symmetric key K_i with each switch S_i in the network. 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 matching rule 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 $(2 \le i \le n)$ 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 store session keys, switches derive them from `VID` on demand. This approach is inspired by [12].

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:

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

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 REV for each packet of the flow. This means that both the source and destination switches should send a verification report for each packet of the flow. When there are multiple flows for verification simultaneously, 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) $\texttt{Verify}(p_i, t_i) = \texttt{true},\ \text{for } \forall i \in [1, m] \Rightarrow$
$\texttt{Verify}(\texttt{Compress}(\{p_i\}_{i=1}^m), \texttt{Compress}(\{t_i\}_{i=1}^m)) = \texttt{true}$
(2) $\texttt{Verify}(p_i, t_i) = \texttt{false},\ \text{for } \exists i \in [1, m] \Rightarrow^{\text{(w.h.p)}}$
$\texttt{Verify}(\texttt{Compress}(\{p_i\}_{i=1}^m), \texttt{Compress}(\{t_i\}_{i=1}^m)) = \texttt{false}$

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 [13], which in turn are based on the Cater-Wagman MAC [14]. Note that homomorphic MACs are designed for defending network coding against pollution attacks [15], 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 $\boldsymbol{p}_F^D \in \mathbb{F}_q^{l+n}$, 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. Also note that each tag depends on the index of the packet, thus each switch should maintain a counter for the flow f.

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:

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

```
Algorithm 1: Verify(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 $f$:
   $\texttt{Path}(\mathcal{R}, f) = (S_0 \to S_1 \to \ldots \to S_{n+1})$;
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;
```

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.

IV. Security Analysis

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

Theorem 1: 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 2 in Appendix. 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}. Since S_{i+1} is not compromised, then according to Theorem 1, the probability that an adversary can forge a tag for S_{i+1}, such that the tag passes the verification, is negligibly larger than $1/q$. Thus, switch bypass can be detected.

Path Detour. Similar to the above case, suppose a packet p is received by a compromised switch S_i, which forwards p

along a path $D_1 \to \ldots \to D_m \to S_{i+1}$. 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, this can still be detected by REV for the following reason. Recall that a tag encodes the last-hop switch identifier, according to Eq (5). Thus, if p is received from D_m instead of S_i, then the tag generated by S_{i+1} will be different and will not pass the verification with high probability. Thus, path detour can be detected.

Out-of-order Traversal. Suppose a packet p has traversed the path $S_0 \to \ldots \to S_n \to S_{n+1}$ in a different order. Then, the number of tags carried by p when reaching S_{n+1} will still be n. However, these tags would be different from what would be generated if the path is traversed in the right order. The reason is that a tag is generated based on not only the hash of p, but also on all tags of p. As a result, even for the same switch and the same packet, the generated tag will be different if the order that the switch appears is different. 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 $S_0 \to S_1 \to \ldots \to S_m$, where S_m is the switch where p is dropped. To verify the enforcement of such kind of ACL rules, it is required that any switch that drops a packet should also send a verification report, as in the packet-level REV. 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. In this sense, the dropping switch takes the role of a destination switch. 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.

Packet Replay. Suppose a compromised switch records flows of previous verification sessions, and replay them into the network. The goal of the adversary is to let these replayed flows pass the controller's verification. In this case, the destination switches would send verification reports to the controller for these flows. However, since the controller has not received any verification reports of these flows from any source switches, it will just forgo the verification reports from the destination switches without verification. Even the replayed packets cannot pass the controller's verification, the adversary may still wish to drain a destination switch's memory by replaying a lot of flows (the destination switch should maintain a flow-packet for each flow). In the proposed REV methods, each packet carries a `TimeStamp`, and switches will not compute tags or compress packets with out-of-date `TimeStamp`. This can to some extent mitigate the above problem.

V. Implementation

REV header format. The REV header sits in-between the IP and TCP/UDP header, consisting of `VID`, `TimeStamp`, `PktHash`, which are all 16-bit long. In addition, there are a variable number of t-Byte tags, each of which is computed by a switch along the packet's path. Note that the current prototype does not include the signature $\sigma_{rK}(\texttt{VID}||\texttt{TimeStamp})$, which is left for future implementation.

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. Both AES and SHA1 are implemented using OpenSSL [16]. Computation over Galois field is performed using the fast Galois field arithmetic library [17], and a multiplication table is pre-computed for fast multiplications.

REV Server. The REV server is an application running atop the Ryu controller [18]. It 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 [11]. The server exposes a REST API that allows users 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/
```

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 server and switches.

REV Datapath. The REV datapath is implemented based on the CPqD OpenFlow software switch [19]. Two extra components are added: (1) a REV agent which communicates to the server with `experimenter` messages, 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 packet counter of a verification flow 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 evaluates the flow-leve REV in terms of bandwidth overhead, processing throughput, and processing delay.

A. Dataset and Setup

Dataset. The dataset is a sample trace of a backbone router, collected in 2011 [20]. 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.

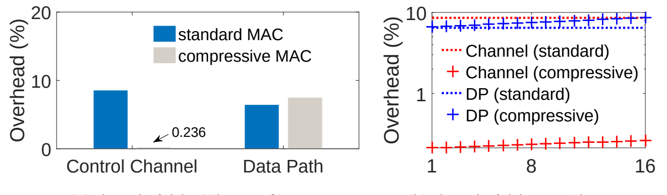
Setup. The evaluation uses a linear topology consisting of one source switch, n core switches, and one destination switch. We generate a flow of f packets, each of which has p Bytes, and feed all these packets to the source switch. Then, the destination switch outputs a compressed hash and n compressed tags. Finally, we let the REV server verify the compressed hash and tags, and record the time t. In the experiments, we measure the bandwidth overhead, processing throughput, and processing delay, by varying the parameters f, p, and n. When we refer to the *trace data*, we mean the specific configuration of $f = 40$ and $p = 757$ Bytes. All experiments on a Linux server with a 3.6GH Intel i7 processor and 32GB memory.

B. Bandwidth Overhead

First consider the flow-level REV which uses the compressive MAC. On the datapath, each packet carries `VID`, `PktHash`, `TimeStamp`, each of which is 16-Byte long, a 1-Byte hop count, and up to n tags, where n is the number of hops. As each tag has t Bytes, the bandwidth overhead per packet on the datapath is then $nt + 49$ Bytes. For the control channel, the source switch should send `VID` and the compressed hash to the controller, while the destination should send `VID`, the compressed hash and n compressed tags. In sum, the total bandwidth overhead per flow on the control channel is $32 + (nt + 32) = nt + 64$ Bytes.

For comparison, consider the packet-level REV which uses standard MACs. In this case, only one tag is needed for each packet, and the bandwidth overhead per packet on the datapath is $t + 48$ 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 $(t + 64)f$ Bytes.

Fig. 2(a) shows the bandwidth overhead for the trace data, when there are 8 hops. We can see that the compressive MAC reduces the bandwidth cost of the control channel by around 97%, compared with standard MACs. The bandwidth overhead of the compressive MAC is slightly higher than that of standard MACs on the datapath. Fig. 2(b) shows the relationship between bandwidth overhead and the number of hops. We can see that using the compressive MAC, the bandwidth overhead for both the control channel and datapath increases slowly with the number of hops.



(a) bandwidth (#hops=8) (b) bandwidth vs. #hops

Fig. 2. Bandwidth overhead on the control channel and datapath, when using standard and compressive MAC, respectively. The results are based on a backbone router trace from CAIDA [20].

C. Verification Throughput

This experiment evaluates the verification throughput, defined as the total number of packets f divided by the verification time t.

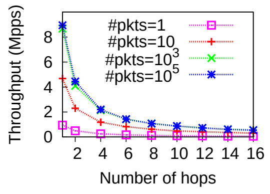
Fig. 3(a) reports the verification throughput for different number of hops. It shows that the throughput drops when the flow path consists of more hops. The reason is that each core switch would append a tag to each packet, and the server needs to verify each of the tags. The throughput is around 1Mpps for the trace data when there are 8 hops.

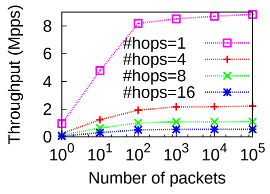
Fig. 3(b) reports the relationship between verification throughput and flow size. We can see that the throughput is rather low (0.12Mpps) when there is only one packet in the flow. This corresponds to the packet-level REV method, where each packet needs to be verified individually. On the other hand, 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.

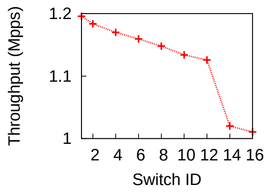
D. Datapath Throughput

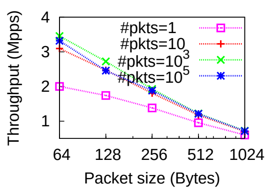
Since CPqD is just a software switch running in Linux user space, it has a rather limited packet forwarding capability. For this reason, CPqD is mainly used to test the functionality of OpenFlow, rather than obtain performance benchmarks. Here, we decide to extract the computation logics of REV as a standalone program, and measure its throughput. We distinguish among three different types of switches: source switches, destination switches, and core switches.

Core switches. Fig. 3(c) 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 tags carried by a packet, the throughput decreases as the switch identifier increases (switch are numbered according to their appearance on the path). 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. Since the core switch processing is bottlenecked by symmetric-key operations, we expect the throughput can be


(a) verification throughput


(b) verification throughput


(c) switch throughput (core)


(d) switch throughput (source)

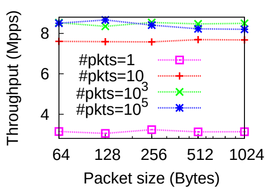
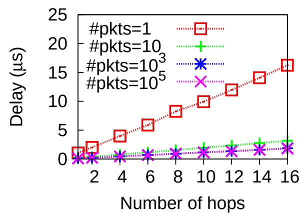

(e) switch throughput (dest.)

Fig. 3. Verification throughput and datapath throughput.


(a) REV server

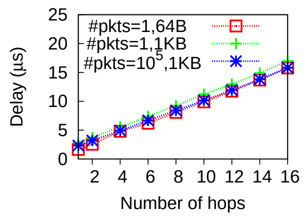

(b) REV datapath

Fig. 4. Delay of the REV server and the REV datapath.

significantly improved by using AES-NI [21], an instruction set for speeding up AES operations.

Edge switches. Fig. 3(d) and Fig. 3(e) 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.

E. Processing Delay

Fig. 4 reports the delay of the REV server and the REV datapath. 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 $3.2\mu s$ to verify a packet when there are 16 hops. The delay for the datapath 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 $16\mu s$ when there are 16 hops. The results show that the flow-level REV incurs minimal latency on the datapath.

VII. Related Work

Verification of SDN data plane. Many dataplane verification tools for SDNs have been proposed recently. Anteater [6], HSA [7], and VeriFlow [8] verify the correctness of dataplane configuration at the controller. However, even the dataplane configuration is correct, switches may still experience faults. ATPG [9] addresses this issue by actively testing the SDN dataplane with probe packets. However, ATPG only checks pairwise reachability, thereby cannot detect faults that do not hurt reachability while changing the forwarding paths. VeriDP [11], [22] addresses this issue by checking whether the packet forwarding behaviors are complying with the dataplane configuration at the controller. However, both ATPG and VeriDP assume switches are trustable, and thus cannot be used for rule enforcement verification.

Path verification. ICING [23] is a path verification mechanism, where each router along the forwarding path computes a MAC for each packet, such that the receiver can check whether a packet has followed the path claimed by the sender. One problem of ICING is that it requires heavy-weight Diffie-Hellman key establishment, and has a high per-packet overhead. OPT [12] achieves a similar goal as ICING, but has a smaller overhead. Both ICING and OPT only support per-packet tagging and verification. When applied to REV, both of them require each packet and tag of a flow be sent to the controller, which will incur a high bandwidth cost.

Message Authentication Codes. Aggregate MACs [24] 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. Homomorphic MACs [13] 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 paths without being modified.

VIII. Conclusion

This paper motivated rule enforcement verification (REV), a new security primitive for software defined networks. In packet-level REV, all switches tagged packets that they forwarded, and exit switches reported the tags to the controller for verification. To reduce the switch-to-controller traffic, this paper proposed flow-level REV based on compressive MAC, a new MAC that allowed tag compression. Emulation showed that by using compressive MAC, the flow-level REV can achieve a 97% reduction in switch-to-controller traffic, and a $8\times$ increase in verification throughput.

Acknowledgement. The author would like to thank all the anonymous reviewers for their valuable comments. This work is supported by the National Key Research and Development Program of China (No. 2016YFB0800100) and the National Natural Science Foundation of China (No. 61402357 and 61572278).

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM CCR*, vol. 38, no. 2, pp. 69–74, 2008.

[2] G. Pickett, "Staying persistent in software defined networks," in *Black Hat Briefings*, 2015.

[3] M. Antikainen, T. Aura, and M. Särelä, "Spook in your network: Attacking an sdn with a compromised openflow switch," in *Secure IT Systems*, 2014.

[4] P.-W. Chi, C.-T. Kuo, J.-W. Guo, and C.-L. Lei, "How to detect a compromised sdn switch," in *IEEE NetSoft*, 2015.

[5] K. Benton, L. J. Camp, and C. Small, "OpenFlow vulnerability assessment," in *ACM HotSDN*, 2013.

[6] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *ACM SIGCOMM*, 2011.

[7] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *NSDI*, 2012.

[8] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *NSDI*, 2013.

[9] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," in *ACM CoNEXT*, 2012.

[10] P. Peresini, M. Kuzniar, and D. Kostic, "Monocle: Dynamic, fine-grained data plane monitoring," in *ACM CoNEXT*, 2015.

[11] P. Zhang, H. Li, C. Hu, L. Hu, and L. Xiong, "Stick to the script: Monitoring the policy compliance of SDN data plane," in *ACM/IEEE ANCS*, 2016.

[12] T. H.-J. Kim, C. Basescu, L. Jia, S. B. Lee, Y.-C. Hu, and A. Perrig, "Lightweight source authentication and path validation," in *ACM SIGCOMM*, 2014.

[13] S. Agrawal and D. Boneh, "Homomorphic MACs: MAC-based integrity for network coding," in *ACNS*, 2009.

[14] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," in *ACM Symposium on Theory of Computing*, 1977.

[15] P. Zhang, Y. Jiang, C. Lin, H. Yao, A. Wasef, and X. S. Shen, "Padding for orthogonality: Efficient subspace authentication for network coding," in *IEEE INFOCOM*, 2011.

[16] "The OpenSSL project," http://www.openssl.org/.

[17] "Fast Galois field arithmetic library in C/C++," http://web.eecs.utk.edu/~plank/plank/papers/CS-07-593.

[18] "The Ryu OpenFlow Controller," https://osrg.github.io/ryu/.

[19] "CPqD: The OpenFlow 1.3 compatible user-space software switch," http://cpqd.github.io/ofsoftswitch13/.

[20] "Packet traces from wide backbone," http://mawi.wide.ad.jp/mawi/samplepoint-F/2011/201101231400.html.

[21] S. Gueron, "Intel Advanced Encryption Standard (AES) New Instructions Set," *Intel*, 2010.

[22] P. Zhang, H. Li, C. Hu, L. Hu, L. Xiong, R. Wang, and Y. Zhang, "Mind the gap: Monitoring the control-data plane consistency," in *ACM CoNEXT*, 2016.

[23] J. Naous, M. Walfish, A. Nicolosi, D. Mazieres, M. Miller, and A. Seehra, "Verifying and enforcing network paths with ICING," in *ACM CoNEXT*, 2011.

[24] J. Katz and A. Y. Lindell, "Aggregate message authentication codes," in *Topics in Cryptology–CT-RSA*, 2008.

Appendix

This appendix presents the definition of *compressive MAC*, a building block of our flow-level REV method.

Definition 3: A (q, l) compressive MAC is defined by three probabilistic polynomial time (PPT) algorithms (**MAC**,**Compress**,**Verify**), where **MAC** generates a tag for a vector of length l defined on the finite field $\mathbb{F}_q$; **Compress** combines multiple vectors and the corresponding tags into a single vector and tag; **Verify** checks the validity of a vector against its tag. The detailed specification is as follows:

- **MAC.** Input: a security key k_M, an identifier id, and a vector $\boldsymbol{p} \in \mathbb{F}_q^l$. Output: a tag t for $\boldsymbol{p}$.
- **Compress.** Input: a security key k_C, n id/vector pairs $\{(id_i, \boldsymbol{p}_i)\}_{i=1}^n$, and their tags $\{t_i\}_{i=1}^n$, where id_i are all distinct. Output: a vector $\boldsymbol{p} = \langle id_i, \boldsymbol{p}_i, k_C\rangle_{i=1}^n \in \mathbb{F}_q^l$, and a tag $t = \langle id_i, t_i, k_C\rangle_{i=1}^n \in \mathbb{F}_q$.
- **Verify.** Input: security key (k_M, k_C), a set of identifiers $\{id_i\}_{i=1}^n$, a vector $\boldsymbol{p}$, and its tag t. Output: 0 (reject) or 1 (accept).

For correctness, we require the above algorithms satisfy:

$$\begin{aligned}&\textbf{Verify}(k_M, k_C, \{id_i\}_{i=1}^n, \langle id_i, \boldsymbol{p}_i, k_C\rangle_{i=1}^n,\\ &\quad\langle id_i, \textbf{MAC}(k_M, id_i, \boldsymbol{p}_i), k_C\rangle_{i=1}^n) = 1\end{aligned}$$

Then, we define the security of compressive MAC via the following game:

Definition 4: Let Ψ=(**MAC, Compress, Verify**) be a (q, l) compressive MAC. Consider a security game **GAME0** played between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$:

- **Setup.** The challenger $\mathcal{C}$ generates a security key pair (k_M, k_C) randomly.
- **Query.** The adversary $\mathcal{A}$ generates n id/vector pairs $\{(id_i, \boldsymbol{p}_i)\}_{i=1}^n$, where id_i are all distinct, and submits them to $\mathcal{C}$. Then, $\mathcal{A}$ adaptively submits queries $(id_i', \boldsymbol{p}_i')$ to $\mathcal{C}$, where id_i' are all distinct. $\mathcal{C}$ returns $t_i' = MAC(k_M, id_i', \boldsymbol{p}_i')$.
- **Output.** $\mathcal{A}$ generates n tuples $\{(id_i^*, \boldsymbol{p}_i^*, t_i^*)\}_{i=1}^n$.

The adversary $\mathcal{A}$ is said to win **GAME0** if:

$$\begin{aligned}&\{id_i\}_{i=1}^n = \{id_i^*\}_{i=1}^n,\ \langle id_i, \boldsymbol{p}_i, k_C\rangle_{i=1}^n = \langle id_i^*, \boldsymbol{p}_i^*, k_C\rangle_{i=1}^n,\\ &\textbf{Verify}(k_M, k_C, \{id_i\}_{i=1}^n, \langle id_i^*, \boldsymbol{p}_i^*, k_C\rangle_{i=1}^n, \langle id_i^*, t_i^*, k_C\rangle_{i=1}^n) = 1,\end{aligned}$$

and either of the following two conditions holds:

1) there exists a $1 \le i \le n$, such that $\boldsymbol{p}_i^* \neq \boldsymbol{p}_i$ (Type I)
2) there exists a $1 \le i \le n$, such that $\mathcal{A}$ has never queried $(id_i, \boldsymbol{p}_i)$ (Type II)

Define the *advantage* MAC-Adv$(\mathcal{A}, \Psi)$ as the probability that $\mathcal{A}$ wins **GAME0** with respect to Ψ. Then, we have the following security definition for compressive MAC:

Definition 5: A (q, l) compressive MAC scheme Ψ is secure if for all PPT adversary $\mathcal{A}$, its advantage MAC-Adv$(\mathcal{A}, \Psi)$ is negligible.

Note we require that the set of identifiers $\{id_i\}_{i=1}^n$ at the sender should be the same as the set of identifiers $\{id_i^*\}_{i=1}^n$ outputted by $\mathcal{A}$. We also require that the compressed packet $\langle id_i, \boldsymbol{p}_i, k_C\rangle_{i=1}^n$ should be the same as $\langle id_i^*, \boldsymbol{p}_i^*, k_C\rangle_{i=1}^n$ that is outputted by $\mathcal{A}$. These requirements can be ensured by incrementing the identifiers sequentially, and letting both the sender and receiver report the compressed packet to the verifier.

Construction 1: Our construction of compressive MAC Ψ uses a pseudorandom generator G and a pseudorandom function F, and is defined as follows.

- **MAC.** Given the security key $k_M = (k_1, k_2)$, a vector $\boldsymbol{p} \in \mathbb{F}_q^l$, and the identifier id, do the following: (1) $\boldsymbol{\alpha} \leftarrow G(k_1) \in \mathbb{F}_q^l$; (2) $\gamma \leftarrow F(k_2, id) \in \mathbb{F}_q$; (3) $t \leftarrow \boldsymbol{\alpha} \cdot \boldsymbol{p} + \gamma \in \mathbb{F}_q$; (4) Output t.
- **Compress.** Given $K_C = k_3$, n id/vector pairs $\{(id_i, \boldsymbol{p}_i)\}_{i=1}^n$, and tags $\{t_i\}_{i=1}^n$, do the following: (1)

$\beta_i \leftarrow F(k_3, id_i) \in \mathbb{F}_q$ for each i; (2) $\boldsymbol{p} \leftarrow \sum_{i=1}^n \beta_i \boldsymbol{p}_i \in \mathbb{F}_q^l$; (3) $t \leftarrow \sum_{i=1}^n \beta_i t_i \in \mathbb{F}_q$; (4) Output $\boldsymbol{p}$ and t.

- **Verify.** Given the security key $(k_M, k_C) = (k_1, k_2, k_3)$, a vector $\boldsymbol{p}$, its tag t, and a set of identifiers $\{id_i\}_{i=1}^n$, do the following: (1) $\boldsymbol{\alpha} \leftarrow G(k_1) \in \mathbb{F}_q^l$; (2) $\beta_i \leftarrow F(k_3, id_i) \in \mathbb{F}_q$; (3) $\gamma \leftarrow \sum_{i=1}^n \beta_i \cdot F(k_2, id_i) \in \mathbb{F}_q$; (4) $t' \leftarrow \boldsymbol{\alpha} \cdot \boldsymbol{p} + \gamma \in \mathbb{F}_q$; (5) Output 1 if $t' = t$, or 0 otherwise.

The correctness of Ψ follows since:

$$\begin{aligned} t' = \boldsymbol{\alpha} \cdot \boldsymbol{p} + \gamma &= \boldsymbol{\alpha} \cdot \sum_{i=1}^n \beta_i \boldsymbol{p}_i + \sum_{i=1}^n \beta_i \gamma_i \\ &= \sum_{i=1}^n \beta_i (\boldsymbol{\alpha} \cdot \boldsymbol{p}_i + \gamma_i) = \sum_{i=1}^n \beta_i t_i = t \end{aligned}$$

Theorem 2: Our construction of compressive MAC Ψ is secure, assuming F and G are secure pseudorandom function and generator, respectively. In specific, for all PPT adversary $\mathcal{A}$, there exists a PRF adversary $\mathcal{B}_1$ and a PRG adversary $\mathcal{B}_2$ whose advantage in wining their respective game is PRF-Adv$(\mathcal{B}_1, F)$ and PRG-Adv$(\mathcal{B}_2, G)$, such that:

$$\text{MAC-Adv}(\mathcal{A}, \Psi) \leq \text{PRF-Adv}(\mathcal{B}_1, F) + \text{PRG-Adv}(\mathcal{B}_2, G) + \frac{1}{q}$$

Proof: Define another two games **GAME1** and **GAME2** as follows. **GAME1** is identical to **GAME0**, except that the output of G is replaced by a truly random element of $\mathbb{F}_q^l$. That is, before responding to MAC queries, the challenger randomly samples $\boldsymbol{\alpha} \xleftarrow{R} \mathbb{F}_q^l$, which is then used in step (1) of **MAC**. **GAME2** is identical to **GAME1**, except that the output of F is replaced by a truly random element of $\mathbb{F}_q$. That is, when responding to each MAC query, the challenger derives $\gamma \xleftarrow{R} \mathbb{F}_q$ in step (2) of **MAC**.

Let $W0$, $W1$, $W2$ be the event that $\mathcal{A}$ wins **GAME0**, **GAME1**, and **GAME2**, respectively. Then, we have:

$$\Pr(W0) = \text{MAC-Adv}(\mathcal{A}, \Psi) \tag{6}$$

In addition, there exists a PRG adversary $\mathcal{B}_2$ such that:

$$\Pr(W0) - \Pr(W1) \leq \text{PRG-Adv}(\mathcal{B}_2, G) \tag{7}$$

, and a PRF adversary $\mathcal{B}_1$ such that:

$$\Pr(W1) - \Pr(W2) \leq \text{PRF-Adv}(\mathcal{B}_1, F) \tag{8}$$

The following defines how the challenger $\mathcal{C}$ interacts with the adversary $\mathcal{A}$ in **GAME2**:

Initialization. $\mathcal{C}$ generates a random vector $\boldsymbol{\alpha} \xleftarrow{R} \mathbb{F}_q^l$. $\mathcal{A}$ submits $\{(id_i, \boldsymbol{p}_i)\}_{i=1}^n$ to $\mathcal{C}$.

Query. $\mathcal{A}$ queries $(id_i', \boldsymbol{p}_i')$. For each i, $\mathcal{C}$ generates $\gamma_i' \xleftarrow{R} \mathbb{F}_q$, and returns $t' = \boldsymbol{\alpha}\boldsymbol{p}_i' + \gamma_i'$.

Output. $\mathcal{A}$ outputs n id/vector/tag tuples $\{(id_i, \boldsymbol{p}_i^*, t_i^*)\}_{i=1}^n$. $\mathcal{C}$ performs the following steps: (1) for each id_i, if $id_i = id_j'$ for some j, then $\gamma_i = \gamma_j'$, else $\gamma_i \xleftarrow{R} \mathbb{F}_q$; (2) for each id_i, generate $\beta_i \xleftarrow{R} \mathbb{F}_q$; (3) check whether the following two equations holds:

$$\sum_{i=1}^n \beta_i \boldsymbol{p}_i = \sum_{i=1}^n \beta_i \boldsymbol{p}_i^* \tag{9}$$

$$\boldsymbol{\alpha} \sum_{i=1}^n \beta_i \boldsymbol{p}_i^* + \sum_{i=1}^n \beta_i \gamma_i = \sum_{i=1}^n \beta_i t_i^* \tag{10}$$

In the following, we are going to compute $\Pr(W2)$. Let T be the event that Type I break happens, and let $\bar{T}$ be the complement event of T. First, we evaluate $\Pr(W2 \wedge T)$, *i.e.*, the probability that $\mathcal{A}$ wins **GAME2** via Type I break. Suppose $\boldsymbol{p}_i^* \neq \boldsymbol{p}_i$ for some i. Then, they must differ in at least one position j, *i.e.*, $\boldsymbol{p}_i^*(j) \neq \boldsymbol{p}_i(j)$. Define set I_j as all k's that satisfy $\boldsymbol{p}_k^*(j) \neq \boldsymbol{p}_k(j)$. Then, combining Eq(9) and the fact that $\boldsymbol{p}_i^*(j) = \boldsymbol{p}_i(j)$ for $\forall i \notin I$, we have:

$$\sum_{i \in I_j} \beta_i \cdot (\boldsymbol{p}_i^*(j) - \boldsymbol{p}_i(j)) = 0 \tag{11}$$

However, since the vector $(\beta_i, id_i \in I_j)$ appears as random vector to $\mathcal{A}$, thus the probability that the nonzero vector $(\boldsymbol{p}_i^*(j) - \boldsymbol{p}_i(j), i \in I_j)$ evaluates the random vector $(\beta_i, i \in I)$ to zero is exactly $1/q$. Thus, $\Pr(W2 \wedge T) = (1/q) \cdot \Pr(T)$.

Then, we evaluate $\Pr(W2 \wedge \bar{T})$, *i.e.*, the probability that $\mathcal{A}$ wins **GAME2** via Type II break. Without loss of generality, assume $\mathcal{A}$ has queried $(id_i, \boldsymbol{p}_i)$ for all $1 \leq i \leq n-1$, but has not queried $(id_n, \boldsymbol{p}_n)$. Then, we have the follow two equations:

$$\boldsymbol{\alpha} \sum_{i=1}^{n-1} \beta_i \boldsymbol{p}_i + \sum_{i=1}^{n-1} \beta_i \gamma_i = \sum_{i=1}^{n-1} \beta_i t_i \tag{12}$$

$$\boldsymbol{\alpha} \sum_{i=1}^{n} \beta_i \boldsymbol{p}_i + \sum_{i=1}^{n} \beta_i \gamma_i = \sum_{i=1}^{n} \beta_i t_i^* \tag{13}$$

By subtracting Eq(12) from Eq(13), we have:

$$\beta_n \boldsymbol{\alpha} \boldsymbol{p}_n + \beta_n \gamma_n = \sum_{i=1}^n \beta_i t_i^* - \sum_{i=1}^{n-1} \beta_i t_i \tag{14}$$

Since $\beta_n \neq 0$, we have:

$$\boldsymbol{\alpha} \boldsymbol{p}_n + \gamma_n = \sum_{i=1}^{n-1} \frac{\beta_i}{\beta_n}(t_i^* - t_i) + t_n^* \tag{15}$$

There are two cases: (1) $\mathcal{A}$ has never queried id_n; (2) $\mathcal{A}$ has queried id_n for $(id_n, \boldsymbol{p}_n')$ with $\boldsymbol{p}_n' \neq \boldsymbol{p}_n$. For case (1) we know γ_n is a random value, and thus the left side of the second line of Eq(15) is a random number, and this equation holds with probability of $1/q$. Now, consider case (2), where we have:

$$\boldsymbol{\alpha} \boldsymbol{p}_n' + \gamma_n = t_n' \tag{16}$$

By substracting Eq(16) from Eq(14), we have

$$\boldsymbol{\alpha}(\boldsymbol{p}_n - \boldsymbol{p}_n') = \sum_{i=1}^{n-1} \frac{\beta_i}{\beta_n}(t_i^* - t_i) + t_n^* - t_n' \tag{17}$$

The right side of the equation is a fixed value, and $\boldsymbol{p}_n - \boldsymbol{p}_n' \neq 0$. Since from $\mathcal{A}$'s perspective, $\boldsymbol{\alpha}$ appears as a random vector, then this equation holds with probability $1/q$. Combining case (1) and (2), we have $\Pr(W2 \wedge \bar{T}) = (1/q) \cdot \Pr(\bar{T})$. Finally, we have:

$$\begin{aligned} \Pr(W2) &= \Pr(W2 \wedge T) + \Pr(W2 \wedge \bar{T}) \\ &= (1/q)(\Pr(T) + \Pr(\bar{T})) = 1/q \end{aligned} \tag{18}$$

Combining Eqs(6)(7)(8)(18), we proved Theorem 2. ■