Detecting Manipulation to Table Rules in the Programmable Data Planes

Ranjitha K*, Karuturi Havya Sree*†, Devansh Garg*, Stavan Nilesh Christian‡, Dheekshitha Bheemanath*†, Rinku Shah§ and Praveen Tammana*
*IIT Hyderabad, India. ‡New York University, USA. §IIIT-Delhi, India.

***Abstract*—Network management systems built on programmable data planes enhance network performance, security, and reliability. However, these systems also increase the attack surface and, hence, are vulnerable to attacks not seen before. We focus on a problem that stems from the fact that a switch data plane trusts the control messages (e.g., table update messages) from upper layers in the switch control plane (OS, SDK, drivers). Since these control messages define the packet forwarding behavior, unauthorized modification to the messages by an adversary at any of these layers can lead to poor performance, privacy compromise, security bypass, and network outage.**

In this paper, we present P4TVal, a P4 Table rule Validation system that detects unauthorized updates to table rules. Our key idea is early detection, where a table rule update is promptly followed by an authenticated rule validation. To realize this idea, we craft and send custom packets to hit specific table rules in the data plane and validate the table rules actually applied to them. We prototype P4TVal for Intel Tofino and BMV2 and demonstrate how P4TVal can detect unauthorized modifications to table rules. Our evaluation shows that P4TVal's early detection reduces the detection time by 10X compared to the state-of-the-art approach.

***Index Terms*—programmable data plane security, securing table update, detecting forwarding table manipulation attack.**

I. Introduction

High-speed programmable Data Planes (PDPs) (*e.g.*, smartNICs [1], switch [2]) and domain-specific network languages (e.g., P4 [3], NPL [4]) have opened up a wide range of opportunities to solve network management problems considered difficult and complex in traditional closed and fixed ASIC-based data planes. They enable faster development of novel network protocols, network function acceleration [5], [6], and in-network computing [7].

Leveraging the PDP capabilities, many data-driven fast control-loop systems ([5], [7]–[15]) improve network performance by quickly adapting to the dynamic network events (*e.g.*, congestion, failures, intrusion detection). These fast control-loop systems execute their data plane logic on network traffic by applying table rules and by updating state in the data plane. More specifically, they perform the following tasks: (a) maintain network statistics in the data plane memory, (b) analyze the statistics, and (c) take action either by updating switch table rules, state in the switch memory, or by configuring flow rate. To keep decision-making fast, some systems implement all three tasks entirely in the data plane, and some of them move analysis to the control plane at the cost of slow decision-making.

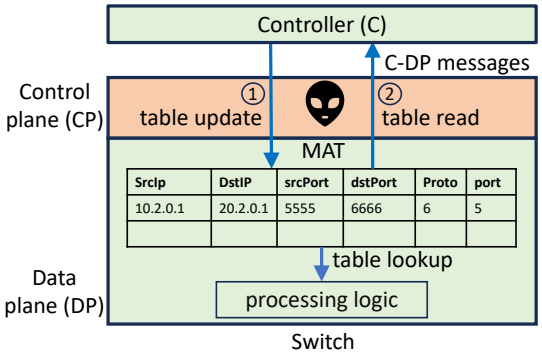


Fig. 1. **Threat model**

Though fast control-loop systems provide better performance and security, they also open up a wide range of new attacks never seen before. Third-party switch operating systems often have exploitable vulnerabilities [16], [17], exposing the switch control plane to various attacks. Many recent works [18]–[20] describe various threat models and attack targets in P4-based data plane systems. In this work, we consider an adversary at the switch control plane (CP) as shown in Fig. 1 who intercepts and modifies the *table update* (install/update/delete) control message from the controller (C), where the message alters the packet forwarding behavior by updating match-action table (MAT) rules in the switch data plane (DP). The adversary can also modify the report sent by the DP in response to a *table read* control message from the controller. Such a Man-in-the-Middle (MitM) adversary can also make unauthorized modifications to data plane table rules (table rule modification attack), causing forwarding anomalies. The adversary can also hide these modifications from being detected by the controller by modifying the table read response.

Attacks on in-network systems tailor-made for enhanced performance and security can hinder the very purpose of the system. For instance, P4Knocking [6] offloads host-based port knocking to the network for enhanced performance. However, [19] shows that P4Knocking is subject to the MitM table rule modification attack discussed above. By modifying the table rule, the MitM attacker bypasses security, defeating the very purpose of the system. Similarly, fast reroute systems (*e.g.*,

† Work done while at IIT Hyderabad

ISBN 978-3-903176-72-0

Blink [9], RouteScout [21]) designed to do performance-aware routing for enhanced performance are also subject to such MitM attacks. The attacker at the CP can modify the table update control messages from the controller so that the traffic is routed to a non-optimal path. This adversely affects the performance, defeating the purpose of the system.

Existing solutions to prevent or detect unauthorized modifications to control messages are not sufficient or too slow to detect the attack. Control messages exchanged between the C-DP (controller and the switch data plane) following P4Runtime [22] are SSL/TLS protected. However, this protection is insufficient if the attacker resides in the CP. Key-based authentication schemes [23]–[26] are designed to run completely in the DP, so they protect updates to DP state that originate from another DP. However, as per the P4 standard [3], the table updates do not happen from the DP; the DP can only look up tables, but not update table rules. Hence, authenticating control messages from the DP is not sufficient. Other detection approaches, such as active probing [27], [28] and distributed statistics collection [29], [30], perform periodic monitoring either by sending active probes or by collecting statistics. A closely related work, REV [31], considers a threat model similar to one shown in Fig. 1. It uses a packet path validation approach, where it captures each packet's actual path and compares it with the expected path to find deviation. This requires the packet to traverse the network, collecting the packet path information before the controller identifies any deviation. However, these approaches are too slow to detect the attack (in the order of seconds), allowing multiple data packets to follow the unintended path.

P4TVal. In this paper, we take a step towards quickly detecting unauthorized modifications to data plane table rules (in a few milliseconds). We propose P4TVal, a P4 Table rule Validation system that detects unauthorized modifications to table rules by following an early detection approach where rule validation promptly follows a rule update. More specifically, P4TVal follows two steps: (1) P4TVal performs the table update (*e.g.*, via P4Runtime); (2) validates whether the update is reflected in the table as expected. P4TVal does rule validation by comparing the rule at the DP with that at the controller (more details in §IV). This allows P4TVal to quickly detect table rule modification attacks and reduce the data packets sent on the unintended paths.

Table read control messages are sent by the controller to ensure the consistency of rules between C-DP. Since P4TVal detects unauthorized modifications to table rules before reading them, we do not foresee the attacker modifying the table read responses. Hence, P4TVal does not perform any explicit validation for table reads (more details in §IV-E).

Challenges. We encounter multiple challenges while designing P4TVal. Firstly, given that we only trust the controller and the DP, and only the controller can initiate rule updates, any validation must happen at the controller. However, retrieving the table rule from the DP, as seen by a packet in the DP, for validation at the controller is challenging. This is because of the computational and resource constraints of PDPs (details in §IV-B). Secondly, table rules may comprise wildcard matches, such as Longest Prefix Match (LPM), and range matches. Validation of these rules is non-trivial as multiple packets can match with a single rule (details in §IV-C). Thirdly, the MitM adversary at the CP can evade rule validation (details in §IV-D).

Key contributions. The key contributions of this paper are:

- We motivate the need for protecting in-network systems from adversaries making unauthorized modifications to data plane table rules (§II).
- We design a detection system that validates the table rules installed in the data plane (§IV).
- We develop a prototype of P4TVal for two switch targets: BMV2 software switch [32] and Intel Tofino [2]. We evaluate P4TVal prototype's effectiveness by detecting attacks on one example in-network system, P4Knocking [6] (§VII).
- We implement a closely related work, REV [31], in BMV2 and compare P4TVal's early detection approach against REV's packet path validation approach. We also evaluate P4TVal overhead in terms of impact on CP and DP performance, and hardware resource utilization (§VIII).

II. Threat Model

Attacker goals. An attacker at a compromised switch OS alters data plane packet forwarding behavior, causing forwarding anomalies in the network by modifying the table update control message from the controller. The attacker also alters the report sent to the controller from the data plane as part of table read control message. This work aims to detect such modifications and thereby minimize the impact of adversarial manipulations on data plane packet forwarding.

Attacker capabilities. We consider an adversary who achieves a compromised switch by installing backdoor applications [16], [19], [31], [33]–[35] either by exploiting the switch OS vulnerabilities or by malware infection or by social engineering a benign operator or by a malicious insider. More specifically, an adversary may exploit stack buffer overflow vulnerability [36] and perform remote code execution [16]. In another case, a malware-infected network administrator system gains access to one of the network switches via keylogging. Then, the malware establishes a reverse connection to the adversary server, installs a backdoor binary onto the switch, and becomes the pivot for the backdoor binary [17]. This allows the adversary's remote server to access an active shell on the compromised switch. The access can be a root or a normal user, depending on the stolen credentials.

On achieving remote shell access to the switch as described above, the adversary can then install a backdoor application (*e.g.*, LD_PRELOAD trick [37]) which preloads a malicious library that can modify the parameters of function calls between the gRPC server agent (P4Runtime server) and the SDK APIs or the driver [19] to update (add/delete/modify) table rules. The adversary can also falsify the table read reports generated by

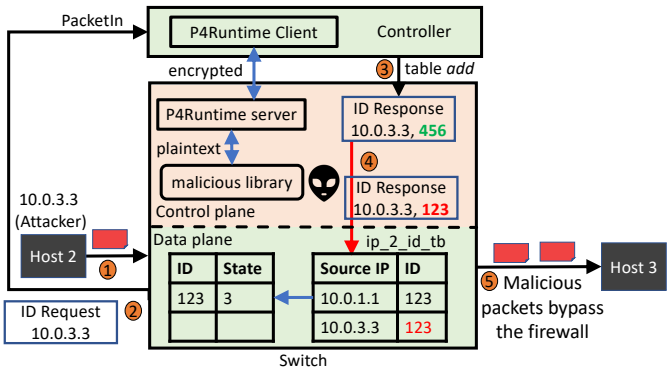


Fig. 2. **Attack on P4Knocking system**

the switch data plane. Note that the adversary can only modify the function call parameters of an ongoing control message processing. We do not consider crafted table updates injected by the adversary because initiating the low-level driver calls is non-trivial, as this requires compromising the gRPC channel.

To summarise, we trust the controller and the switch data plane (DP), but not the switch control plane (CP). The MitM adversary at the CP can modify the table update/read message from/to the controller.

Example table rule modification attack on P4Knocking system. Port knocking is an authentication mechanism used to hide services from unauthorized users, thereby avoiding undesired connection trials. A service port appears closed until the user generates a connection attempt on a pre-configured set of closed ports and gets access.

Port knocking in P4. P4Knocking [6] is a P4 implementation of port knocking that allows seamless deployment of firewall functions in the data plane. Access to a service port is allowed only if the host requesting access sends a series of packets with the correct sequence of TCP destination port numbers (secret knock sequence). P4Knocking implements this using a P4 register by storing an integer from 0 to 3, indicating the number of consecutive packets received from a given host (IP address) that satisfy the secret knock sequence. Once the register value for an IP address is equal to 3, then the traffic from that IP address is allowed to pass through the firewall. To reduce memory footprint, P4Knocking uses an optimization where a 16-bit ID is used to index the register array instead of 32-bit IP addresses. These IDs are assigned by the controller on receiving a PacketIn from the DP. This ID to IP address mapping is maintained in a table *ip_2_id_tb*.

Attack. Black et al., in their work [19], demonstrate attacking P4Knocking by editing the arguments of the table update (add) control message while inserting an entry to *ip_2_id_tb* table (Fig. 2). They modified the entry so that the attacker's IP address gets mapped to an already existing ID, thus allowing the attacker's traffic to bypass the firewall.

III. Requirements and Related Works

An ideal solution to the MitM attack making unauthorized modifications to data plane rules (discussed in §II) is to prevent the attack in the first place; however, programmable data plane constraints make it non-viable. Preventing the attack requires updates to table rules from the DP because, according to our threat model, we only trust the DP and not the CP (§II). However, as per the P4 language standard, ingress packets (or custom messages) cannot update table entries. More specifically, while processing a packet, the DP can look up table entries, but updates to table entries are not allowed; only the CP (OS, SDK, or drivers) can update table entries. This decision was taken by the community considering the complexity involved in implementing the ability to modify tables in the data plane [38]. This constraint makes attack detection the only viable solution.

A. Requirements

Given the specific design purpose of in-network systems and the constraints discussed above, we present the following requirements to secure table rule updates.

[R1] Early detect. Attack must be detected as early as possible such that its impact is negligible. More specifically, the time to detect the attack should be near real-time such that the impact of the attack in terms of the number of data packets compromised on the unintended path is negligible.

[R2] Single source-based detection. The detection system must be dependent only on the information received from a single data plane. Independent of other switches in the network, the detection system must accurately detect the MitM attack on any particular switch. This is necessary as in-network systems tailor-made for specific purposes are often deployed as standalone systems.

[R3] Robust. Compromising the detection system must be hard for the adversary; the detection system must be robust towards attacks from the MitM adversary at the CP.

B. Related works and research gap

P4 data planes are vulnerable. Prior work [18]–[20], [34], [39] highlights how programmability and statefulness make P4 data planes vulnerable to various security attacks. Agape et al. [34] present the security landscape and characterize the attack surface of p4-based SDN systems. Authors of [18], [19] provide an elaborate analysis of P4-SDN architecture and showcase possible attacks that alter the switch's forwarding behavior on multiple data plane systems. [39] shows how buggy P4 programs can be exploited to attack the data plane. [20] highlights common design pitfalls in switch-based applications due to resource-constrained programming, making system design vulnerable to adversarial exploitation. We complement these works by taking a step towards securing programmable data planes from manipulation of packet forwarding behavior.

Prevention approaches using authentication in the data plane are insufficient. Multiple works [23]–[26] suggest securing communication between two data planes by applying authentication, integrity, and encryption. They prevent unauthorized modifications to data plane states by authenticating control messages in the DP. Their approach only considers control messages modifying the data plane state (register

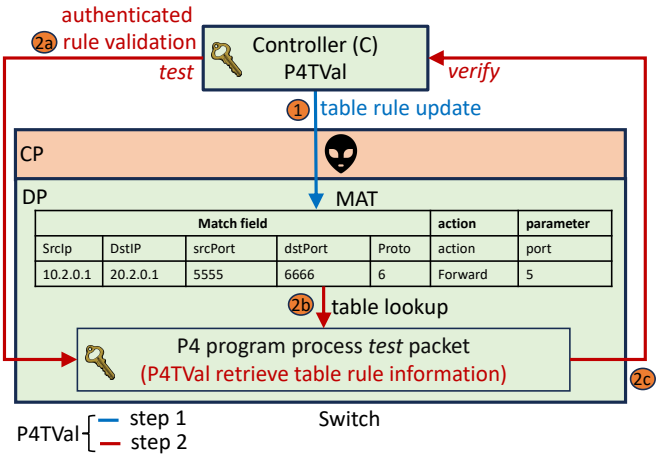


Fig. 3. **Detecting unauthorized table rule modifications using P4TVal**

read/write); they do not consider control messages that update table rules. As per the P4 language standard, DP can only look up table entries; updates to table entries are not allowed. Hence, solutions to authenticate control messages in the data plane do not work for table update control messages, failing to meet R1. Our work complements these works by securing table update control messages.

Detection approaches based on packet path tracking are slow. Packet path validation approaches [31], [40] and probe-based approaches [27], [28], designed for securing SDN-OpenFlow data plane, ensure rule enforcement in the switch data plane by validating the path the data packet takes by comparing the packet's actual path with the expected path. This approach detects adversarial manipulations to table updates. However, it takes longer to detect because this requires the data/probe packet to traverse the network and collect the path information before the controller can process and detect the attack. This allows multiple data packets to get compromised on the unintended path. Also, as the size of the network grows, the time to detect increases with increasing number of switches in the network. Moreover, malicious switches in the network can manipulate the path-tracking information in the packet to evade detection, thus failing to meet R1, R2, and R3.

Distributed monitoring approaches compromise accuracy with malicious switches in the network. Statistics-based approaches [29], [30], [33], [41] use distributed statistics collected periodically from multiple switches to find forwarding anomalies; it takes tens of seconds to detect the anomaly, thus failing to meet R1. Also, these works assume that a majority of switches in the network are benign. However, malign switches can misreport statistics, adversely affecting the detection accuracy and thus failing to meet R2, and R3.

IV. DESIGN

A. Overview

We design P4TVal, a system to detect unauthorized modifications to data plane table rules. We achieve this by validating the table rules after the controller updates them. This takes us to the question *"how to validate the data plane table rules?"*.

Digest-based validation is slow and has high overheads. One approach is to validate the rules from the DP. To achieve this, we can add an action parameter to the table that holds the digest computed over the rule using a secret key shared between C-DP. On packet arrival, the DP computes the digest for the rule and validates it against the digest present in the rule. However, this approach delays the detection until a data packet hits the rule (thus does not hold R1). This approach requires an additional digest field added to each table rule in the DP, increasing the memory footprint. Moreover, an update to the secret key requires updating the digest for all the DP rules, incurring high overhead at the controller. Since the root of trust is at the controller, P4TVal chooses to validate the rules at the controller.

P4TVal. We design P4TVal, a system to validate data plane table rules and detect unauthorized modifications to the rules. Our key idea is to validate the DP table rules from the controller by following an early-detect approach (meets R1). More specifically, as shown in Fig. 3, we allow updating the table rule by the controller (step 1), which is immediately followed by an authenticated rule validation (step 2). In step 2, we send a custom *test* packet to the DP with headers matching the table rule. P4TVal's P4 logic processes the test packet in the data plane, retrieves the table rule information matched with the test packet from DP (meets R2), and shares the information with the controller for validation.

To realize this idea, P4TVal addresses multiple challenges while retrieving rule information and validating the information at the controller.

[C1] Retrieving and sharing the DP rule applied on the test packet. Validating the DP rule at the controller requires P4TVal to retrieve the information of the rule applied on the test packet and share it with the controller. The challenge is retrieving and sharing under programmable data planes' constraints on allowed per-packet operations. We address this by carefully identifying the rule information needed for validation and formatting the response (*i.e.*, *verify* message) that carries the information to the controller (details in §IV-B).

[C2] Validating wildcard rules. Wildcard rules with LPM and range match types match with multiple data packets. For instance, LPM match entry with prefix length N can match to $2^{(32-N)}$ destination address. Sending $2^{(32-N)}$ custom test packets to validate the rule is inefficient. We address this challenge by carefully identifying the modifications the attacker can make and validating the boundaries of such match types (more details in §IV-C).

[C3] Protecting validation from MitM adversary. The MitM attacker at the CP can evade rule validation by compromising the table information retrieved from the DP. We address this challenge by protecting the rule validation using end-to-end authentication between C-DP (meets R3). More details on the authentication process are in §IV-D.

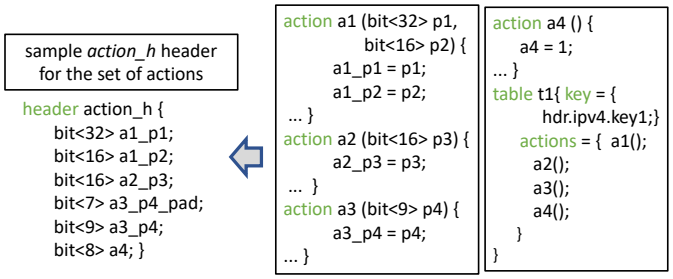


Fig. 4. **Common structure shared between C-DP for retrieving table rule information**

B. Rule validation

Our key idea is to retrieve the table rule information as seen in the DP and compare it against the actuals at the controller. To achieve this, we find answers to two questions:

What information about table rules is needed for validation? A table rule comprises: (1) a match field (key), (2) an action field, and (3) action parameters. The MitM attacker at the CP (as discussed in §II) can modify any of these three parts in a table rule. So, to detect the attack, we need to validate the match, action, and action parameters of the installed table rule with that of the controller.

How do you retrieve the table rule information? To retrieve the table rule information, we send a custom message containing a test packet with headers set to match the rule's match field such that the test packet hits the table rule. Upon hitting the rule, the DP captures the set of actions and action parameters applied to the test packet using a common data structure, *action_h* (as shown in Fig. 4), shared between the controller and the data plane. *action_h* structure has action parameters (as fields) of all actions of every table. To summarize, *action_h* contains the information of the table rule applied on the test packet and returned to the controller, where it is validated.

Validating table rule. To retrieve and validate table rule information, P4TVal uses two messages: (1) *test*, and (2) *verify*. The format of these messages is shown in Fig. 5(a). *hdrType* field specifies one among *test* message, *verify* message or an alert message. *digest* field is used for protecting *test* and *verify* messages from MitM adversary (more details later). *seqNum* maps a response to its corresponding request.

After a rule update, the controller promptly sends a *test* message containing a sample packet with header values that match the table rule to be validated (step 2 in Fig. 3). Currently, we manually construct the test message. Instead, one can automatically generate test packets using packet generation tools like p4pktgen [42]. On receiving the *test* message, DP processes the sample packet in the *test* message like any other data packet and retrieves the actions and associated action parameters applied in the *action_h* header (as shown in Fig. 5(b)). At the end of the processing pipeline, DP constructs and sends the *verify* message (contains *action_h*) to the controller.

On receiving the *verify* message, the controller checks whether the observed actions applied to the sample packet, that is, *action_h* header values are the same as the expected set of actions determined from the table update request in the previous step. For instance, if the table update request adds or modifies an entry, the controller checks whether the observed action parameters and the table entry's action parameters are the same. If the table update request deletes an entry, then it checks whether the observed action parameters are set to either null or the default action, as the sample packet is expected not to hit the deleted entry. If the observed action parameters do not match the expected ones, the controller raises an alert. Note that the mismatch could be due to some bugs in the stack [43] or adversarial manipulation. Nevertheless, P4TVal helps to detect the mismatch such that the network operator can debug the issue further. Additionally, the controller also raises an alert if it fails to receive a *verify* packet in response to the *test* packet.

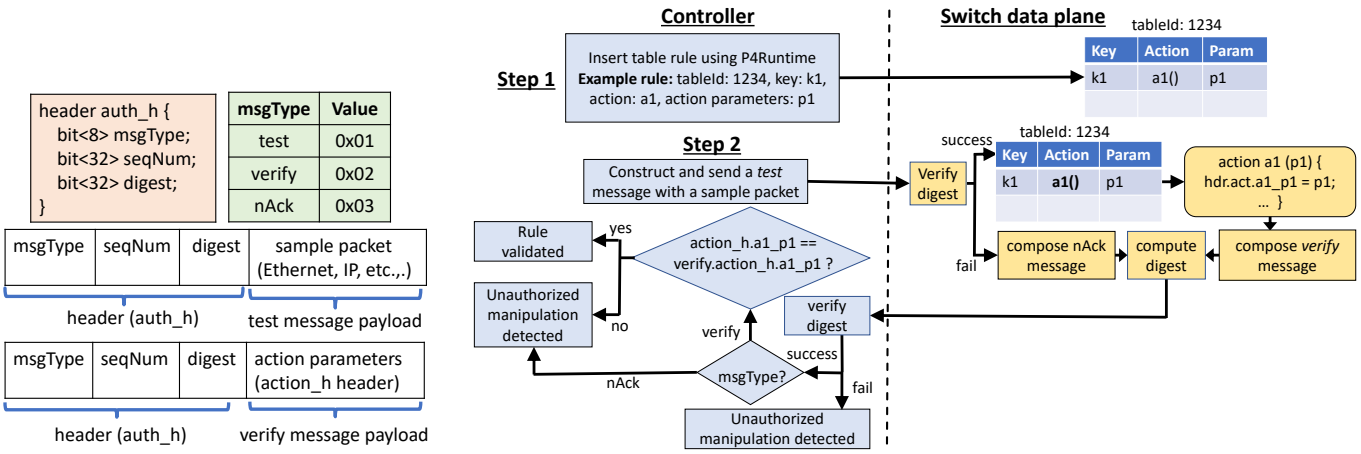
C. Handling different match-types

A table rule may have multiple match fields with one of three types: exact, range, or LPM. A single *test* message is sufficient to validate a table rule with only exact type fields. However, multiple messages are required to validate a table rule with range or LPM type. One key observation is that validating lower and upper bound values in these two match types is sufficient to detect adversarial manipulation.

Handling LPM match. Consider the destination IP address field of the LPM match type where an adversary can modify either prefix or prefix length. To detect any modification to the prefix, a single test message with the destination address the same as the prefix is sufficient. In the case of prefix length modification, an adversary can either increase the prefix length (*i.e.*, decrease the number of IP addresses in a subnet) or decrease the prefix length (*i.e.*, increase the number of IP addresses). In the former case, a test packet with a destination IP address set to the last IP address of the original subnet is expected to hit the user-programmed rule, but it will not; thus, applied actions would differ from the expected one. In the latter case, a test packet with a destination IP address set to the (N-1)th bit of the first IP address of the actual subnet is expected to miss the user-programmed rule, but it will be a hit because the original subnet is a subset of modified larger subnet. In summary, P4TVal requires at least three test messages to validate a table entry of type LPM.

Handling range match. Consider a table entry with range match type, say *range(r1, r2)*. An adversary can modify r1, r2, or both r1 and r2. In this case, we need four test messages with match values set to r1, r2, r1-1, and r2+1. For example, if the range (50,100) is modified to range (60,120) by an adversary, there are four test messages with match values set to 50, 100, 49, and 101. The test packet with a matching key set to 50 is expected to hit the table rule, but it will be a miss; thus, applied actions would differ from the expected one.

To summarize, to validate a table rule with M LPM type fields, N range type fields, and P exact type fields, at most $M*3+N*4+P*1$, test messages are required. For example, if a table rule has one LPM, one range, and one exact match type field, 8 test messages are required to validate the table



(a) **Formats of *test* and *verify* messages.** (b) **Workflow of authenticated table rule validation**

Fig. 5. **P4TVal table rule validation**

rule. It is worth noting that the number of test messages is independent of the prefix length or the range size.

D. Protecting rule validation

The adversary at the CP can modify the *test* and *verify* packet and compromise the detection mechanism. To prevent the adversary from evading detection, P4TVal ensures the integrity and authentication of *test* and *verify* packets completely in the DP. Our requirement to authenticate *test* and *verify* aligns with the objective of existing works that perform key exchange [23] and key-based authentication [26] in the DP. Taking inspiration from these works, we use key-based HMAC to generate message digest and ensure the authenticity and integrity of *test* and *verify* packets. We assume the secret key is already shared between the controller and the DP.

Authenticating *test* message. Before sending the *test* message, the controller computes the digest over two fields: *auth_h* and the packet payload, using the secret key shared between C-DP. On receiving the *test* message, the DP first computes the digest on the same fields using the shared key and compares it with the digest in the *test* message. If the digest verification fails, the DP sends an alert. If the digest matches, DP processes the packet and updates the action parameters as the packet traverses the processing pipeline. At the end of the processing pipeline, the DP constructs the *verify* message with *msgType* (0x02) and *digest* (computed over *action_h* header values, *hdrType*, and *msgType* using the shared secret). On the other hand, if the digests do not match, it sends a *nAck* message with *msgType* set to 0x03 and *digest* computed over *msgType* and *seqNum*.

Authenticating *verify* message. On receiving the *verify* message, the controller performs digest verification to ensure the integrity of the message. On successful verification, the controller validates the table rule as discussed in the above section. If the *verify* message digest verification fails, or the observed action parameters do not match the expected one, the controller raises an alert.

E. Validation of table reads

In general, SDN controllers (*e.g.*, ONOS) report inconsistency [19] whenever there is a mismatch between the logs maintained at the controller and the table rules read. However, an attacker could modify *both* table update requests and read responses such that the controller cannot detect the mismatch. With P4TVal's detection of unauthorized modifications to updates (as described earlier) and the network operator's actions to ensure table rules in the DP and the table entry logs at the controller are the same, any modifications to the table read responses can be detected by the controller.

V. Implementation

This section presents the implementation details of P4TVal prototype developed on BMv2 [32] and Tofino [2] target. We instrument the P4 program for two purposes: (1) to retrieve table rule information from the DP (for validation) and (2) to perform digest computation and verification (for authentication). We added around 70 lines of P4 code. We implement the controller in Python3.7 using approximately 50 lines of code to perform digest computation verification and rule validation.

Retrieving table rule information. To retrieve the table rule information, DP generates *verify* message for every *test* message. For this, we instrument the P4 code with (1) *action_h* structure; and (2) instructions in each action block for copying action parameters to the respective field in *action_h*. At the end of the packet processing pipeline, *action_h* contains all action parameters of the actions applied on a packet as it traverses the switch pipeline. We implement *test* and *verify* packets as PacketOut and PacketIn, respectively.

Digest computation and verification. Code for digest computation and verification is added at the start and end of the ingress pipeline. We used CRC32 as the hash algorithm for digest computation, and the secret key shared between C-DP for digest computation is stored in the register. We limited ourselves to using 32-bit digest as Tofino inherently supports only 32-bit ALUs.

VI. Security Guarantees

In this section, we present the security analysis of P4TVal. P4TVal assumes that the secret key is securely shared between C-DP and no adversary has access to it. With this assumption, we discuss the security guarantees of P4TVal and other possible attacks adversaries can launch against P4TVal.

Digest size. The output digest size is 32 bits. An attacker intending to modify the message can try at most 2^{32} different possibilities for the actual digest corresponding to the message. However, during these adversarial trials of guessing the digest, an alert is raised, indicating a malformed digest encountered at the DP or the controller, revealing the possibility of the adversary. Thus, P4TVal is safe from such brute force attacks for guessing the digest.

Denial-of-service (DoS) attack. A denial-of-service attack can be launched on the controller by manipulating many messages to the controller or DP. There are two possibilities here. First, if an adversary modifies many *test* messages sent to the DP, the DP finds digests do not match and sends a stream of alert messages (one for every message) to the controller. This can jam the C-DP communication link and overwhelm the controller, causing a DoS attack. Second, many modified response messages (*nAck* or *verify*) sent to the controller can also overwhelm the controller and lead to a DoS attack. In both cases, upon detecting unauthorized modification, ideally, the network operator should take appropriate action (*e.g.*, isolate suspicious switch). If no action is taken, it can lead to a DoS attack on the controller, as explained earlier. In this situation, to mitigate the first type of attack, one can set a threshold on the number of alert messages sent to the controller in a specific time period from the DP. To mitigate the second type of attack, the controller should keep a threshold on the difference in the number of requests sent and responses received in a specific time period. Moreover, a list of sequence numbers not yet acknowledged can help to defend such attack.

Replay attack. An adversary may record *test* or *verify* messages and replay them. Since *test* message does not modify table rules, it does not affect DP packet processing behavior. Replaying them can only cause a DoS attack, which can be addressed as explained above.

VII. Usecase

In this section, we demonstrate how an attack on the data plane system, P4Knocking [6], can be detected using P4TVal. We assume that the attacker has privileges as per the threat model discussed in section II.

Detection using P4TVal. We recreated the attack on the P4Knocking system shown in Fig. 2 by modifying functions in the switch device_manager layer such that the table *add* request from the controller is modified (MitM attack). With P4TVal (as shown in Fig. 6), after processing the table *add* request, the controller sends a *test* message with a sample packet which hits the newly inserted rule (*i.e.*, key 10.0.3.3) in the table *ip_2_id_tb*. P4TVal copies the action parameter (*i.e.*, 123) of the rule it hits to *verify* message and sends the same to the controller. On receiving the *verify* message, the controller verifies whether the value received (123) is the same as what it sent (456) in the table add request. Here, the values don't match; thus, the controller detects that the table rule has not been added as expected. Since *test* and *verify* are authenticated using the shared secret key, any attempt by the adversary to modify them will be detected by the DP and the controller, respectively.

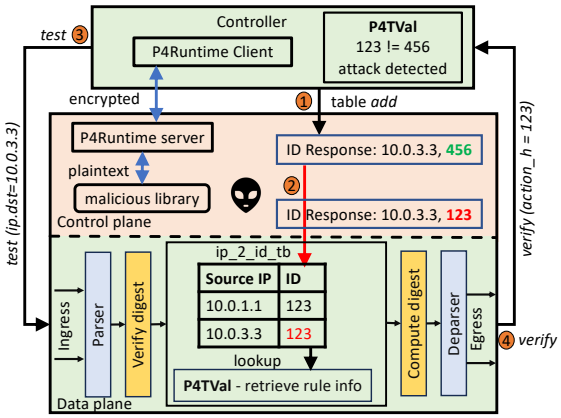


Fig. 6. **Detecting attack on P4Knocking system with P4TVal**

VIII. Evaluation

Our evaluation aims to answer four important questions: (1) How quickly does P4TVal detect table rule modification attack (§VIII-A)?, (2) Does P4TVal impact the performance of the P4 application that runs on a P4 switch target (§VIII-B)?, (3) What is the impact of P4TVal on the control plane performance (§VIII-C)?, and (4) How much additional hardware resource is required to implement P4TVal (§VIII-D)?

Experimental setup. We deployed P4TVal on Aurora 610 Intel Tofino [44] switch that runs on ONL1.4 NOS. We compiled P4TVal using bf-sde 9.9.0, the Intel Tofino's SDE (software development environment). We use a data plane program written in P4 language that performs destination-based layer-3 port forwarding as the *Baseline* for our evaluation. This program comprises two match-action tables and one register implementation. To evaluate the overheads caused by P4TVal, we instrument the baseline program to use P4TVal to detect unauthorized modifications to table rules. We use the packet test framework (PTF) available with Tofino SDE to generate table update message and *test* message and to receive *verify* message.

A. P4TVal early-detect

To study the improvement in attack detection time that P4TVal achieves with the early-detect approach, we compare P4TVal with a closely related work, REV [31]. REV performs rule enforcement by validating the packet's observed path with the expected path. We implemented REV in BMV2 and studied the time REV took to detect the attack. We simulated the attack by modifying the table rule via the switch CLI. We

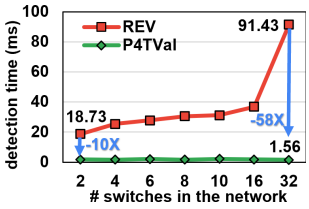


Fig. 7. **Detection time with increasing network size**

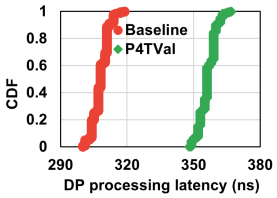


Fig. 8. **CDF of packet processing latency in the data plane.**

TABLE I
IMPACT OF P4TVAL ON TABLE ADD THROUGHPUT AND LATENCY

Program	Avg. throughput (req/sec)	Avg. req. completion time (in ms)
Baseline	1360.14	0.71
P4TVal	585.69	1.77

also increased the number of switches in the network and studied the attack detection time. As depicted in Fig. 7, we observe that P4TVal reduces the attack detection time by 10 times when the network comprises only two switches. As the number of switches increases, the time to detect the attack increases linearly with REV. However, with P4TVal, the time to detect remains constant irrespective of the number of hops as the verification is done with each standalone DP (meeting requirement R3, sole source detection).

B. Impact on data plane performance

To understand the impact of P4TVal on data plane processing time, we sent 5000 IP packets to the DP that runs *Baseline* and *P4TVal* P4 programs. We collected the time spent by each packet in the *Ingress* pipeline of the DP as P4TVal adds additional processing only in the Ingress pipeline. Fig. 8 shows the CDF plot of the processing latency. We observe that P4TVal adds latency of $\sim 50ns$. This is because P4TVal adds a few conditional and assignment operations for authentication and table rule validation, respectively. The number of conditions is constant, but the number of assignment operations increases linearly with the number of tables the packet hits. Thus, scaling to P4 programs with many tables will have an incremental increase in the processing latency.

C. Impact on control plane performance

We study the impact of P4TVal on control plane performance using two metrics: (1) average throughput (table updates completed per sec), and (2) average request completion time. We sent table update control messages sequentially for 30 seconds and reported the average value. Table I shows the throughput and the request completion time for table update requests for *Baseline* and *P4TVal*. We observe that the throughput of table *add* is decreased by $\sim 55\%$ for *P4TVal*, when done sequentially. This decrease is due to the additional validation (*test* packet composition and transmission, followed by verification of *verify* packet) that P4TVal does. We observe (from Table I) that the validation process (relying on the PTF python stack), on average, takes more than a millisecond. It is worth noting that this throughput decrease is reported for the worst case scenario where table *add* followed by validation is done sequentially. However, parallel table rule validation by the controller can further bring down the throughput drop to $\sim 30\%$.

TABLE II
P4TVAL OVERHEAD IN TERMS OF HARDWARE RESOURCE UTILISATION

Program	Hardware Resource			
	TCAM	SRAM	Hash Units	PHV
Baseline	8.3%	2.3%	0%	11.1%
P4TVal	8.3%	2.3%	15.3%	12.5%

TABLE III
PHV UTILIZATION WITH P4TVAL AS WE SCALE THE NUMBER OF ACTION HEADER FIELDS.

#fields in *action_h*	1	2	4	8	16	32	64	80
PHV Util (in %)	12.6	13.3	14.3	16.2	20.1	27.8	43.1	50.8

D. Resource overhead

Table II shows the switch resource utilization for *Baseline* and *P4TVal*. P4TVal adds code for (1) digest computation and verification (*Hash Units*), and (2) *action_h* header definition (*PHV*), and (3) copying the action parameters to *action_h* header in each action block (*PHV*).

P4TVal implementation introduces digest computation and verification that increases the hash unit utilization, but this increment is constant, i.e., the usage does not vary based on the P4 program or network topology. However, P4TVal's codebase grows with the increase in the number of actions and action parameters within a P4 program (see Fig. 4). For instance, a program with M tables, N actions per table, and P action parameters per action requires $M*N*P$ fields in the *action_h* header. The header parameters and values are stored in PHV (packet header vector) containers. As the hardware supports limited PHV containers, this poses a scalability challenge on the number of tables that P4TVal can support. Table III shows that the port forwarding program with P4TVal was able to scale up to 80 action parameters. Note that the PHV containers are distributed among the ingress and egress pipelines, so our P4 program that runs within the ingress pipeline can use up to a maximum of 50% PHV resources.

How can we scale the number of P4 program tables with P4TVal? One of the optimizations could be achieved by replacing the action parameters for each table action (P) with a 32-bit digest. This will reduce the PHV resource usage from $M*N*P$ to $M*N$, i.e., the P4 program tables can scale by $P\times$. This optimization will induce an acceptable amount of increase in Hash Unit usage. Alternatively, path-profiling techniques such as DBVal [43]) make use of Ball-Larus [45] encoding to uniquely identify the table and action executed by a packet. Using such techniques, P4TVal can reduce the number of fields per table to two, (1) to store unique path code and (2) to store the digest of action parameters, thereby reducing the number of header fields to $M*2$. We plan to incorporate these optimizations into P4TVal as future work.

IX. Conclusion

In this work, we motivate the need for detecting adversarial manipulation of packet forwarding behavior in P4 data planes by presenting a threat model and associated attack. We propose P4TVal, a system that detects unauthorized modifications to table rule updates using authenticated rule validation. With an example use case, we show the efficacy of P4TVal in detecting adversarial modifications of control messages. We prototype P4TVal on BMV2 and Tofino targets and present the evaluation of the prototype implemented on the Tofino switch in terms of its impact on control message processing, resource overhead, and data plane packet processing.

X. acknowledgement

We thank the anonymous reviewers for their insightful feedback. We thank Prashanth, Shiv, and Animesh for their feedback and help on the earlier drafts. This work is supported by the National Security Council Secretariat (NSCS), India, and the Prime Minister Research Fellowship (PMRF).

References

[1] (2016) Netronome Agilio CX SmartNICs. [Online]. Available: https://www.netronome.com/products/agilio-cx/

[2] (2020) Intel Intelligent Fabric Processors. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html

[3] (2015) P4 Open Source Programming Language. [Online]. Available: https://p4.org/

[4] NPL. [Online]. Available: https://nplang.org/

[5] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *ACM SIGCOMM, 2017*.

[6] E. O. Zaballa, D. Franco, Z. Zhou, and M. S. Berger, "P4knocking: Offloading host-based firewall functionalities to the network," in *IEEE ICIN, 2020*.

[7] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *ACM SOSP, 2017*.

[8] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *ACM SOSR, 2016*.

[9] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever, "Blink: Fast connectivity recovery entirely in the data plane," in *USENIX NSDI, 2019*.

[10] K.-F. Hsu, R. Beckett, A. Chen, J. Rexford, and D. Walker, "Contra: A programmable system for performance-aware routing," in *USENIX NSDI, 2020*.

[11] Q. Kang, L. Xue, A. Morrison, Y. Tang, A. Chen, and X. Luo, "Programmable in-network security for context-aware BYOD policies," in *USENIX Security*, 2020.

[12] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh *et al.*, "Hpcc: High precision congestion control," in *ACM SIGCOMM, 2019*.

[13] G. Li, M. Zhang, C. Liu, X. Kong, A. Chen, G. Gu, and H. Duan, "Nethcf: Enabling line-rate and adaptive spoofed ip traffic filtering," in *IEEE ICNP, 2019*.

[14] J. Xing, Q. Kang, and A. Chen, "{NetWarden}: Mitigating network covert channels while preserving performance," in *USENIX Security, 2020*.

[15] Z. Liu, H. Namkung, G. Nikolaidis, J. Lee, C. Kim, X. Jin, V. Braverman, M. Yu, and V. Sekar, "Jaqen: A {High-Performance}{Switch-Native} approach for detecting and mitigating volumetric {DDoS} attacks with programmable switches," in *USENIX Security, 2021*.

[16] K. Thimmaraju, B. Shastry, T. Fiebig, F. Hetzelt, J.-P. Seifert, A. Feldmann, and S. Schmid, "Taking control of sdn-based cloud systems via the data plane," in *ACM SOSR, 2018*.

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

[18] R. Meier, T. Holterbach, S. Keck, M. Stähli, V. Lenders, A. Singla, and L. Vanbever, "(self) driving under the influence: Intoxicating adversarial network inputs," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, 2019, pp. 34–42.

[19] C. Black and S. Scott-Hayward, "Adversarial exploitation of p4 data planes," in *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 2021, pp. 508–514.

[20] L. Wang, P. Mittal, and J. Rexford, "Data-plane security applications in adversarial settings," *ACM SIGCOMM Computer Communication Review*, vol. 52, no. 2, pp. 2–9, 2022.

[21] M. Apostolaki, A. Singla, and L. Vanbever, "Performance-driven internet path selection," in *ACM SOSR, 2021*.

[22] P4Runtime. [Online]. Available: https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html

[23] I. Oliveira, E. Neto, R. Immich, R. Fontes, A. Neto, F. Rodriguez, and C. E. Rothenberg, "Dh-aes-p4: on-premise encryption and in-band key-exchange in p4 fully programmable data planes," in *IEEE NFV-SDN, 2021*.

[24] F. Hauser, M. Schmidt, M. Häberle, and M. Menth, "P4-macsec: Dynamic topology monitoring and data layer protection with macsec in p4-based sdn," *IEEE Access*, vol. 8, pp. 58 845–58 858, 2020.

[25] T. Datta, N. Feamster, J. Rexford, and L. Wang, "{spine}: Surveillance protection in the network elements," in *USENIX FOCI, 2019*.

[26] D. Kong, Z. Zhou, Y. Shen, X. Chen, Q. Cheng, D. Zhang, and C. Wu, "In-band network telemetry manipulation attacks and countermeasures in programmable networks," in *IEEE IWQoS, 2023*.

[27] 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 *IEEE INFOCOM, 2016*.

[28] Y.-C. Chiu and P.-C. Lin, "Rapid detection of disobedient forwarding on compromised openflow switches," in *IEEE ICNC, 2017*.

[29] A. Kamisiński and C. Fung, "Flowmon: Detecting malicious switches in software-defined networks," in *Proceedings of the 2015 Workshop on Automated Decision Making for Active Cyber Defense*, 2015, pp. 39–45.

[30] P. Zhang, S. Xu, Z. Yang, H. Li, Q. Li, H. Wang, and C. Hu, "Foces: Detecting forwarding anomalies in software defined networks," in *IEEE ICDCS, 2018*.

[31] P. Zhang, H. Wu, D. Zhang, and Q. Li, "Verifying rule enforcement in software defined networks with rev," *IEEE/ACM ToN*, vol. 28, no. 2, pp. 917–929, 2020.

[32] BMV2 software switch. [Online]. Available: https://github.com/p4lang/behavioral-model

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

[34] A.-A. Agape, M. C. Danceanu, R. R. Hansen, and S. Schmid, "Charting the security landscape of programmable dataplanes," *arXiv preprint arXiv:1807.00128*, 2018.

[35] A. Shaghaghi, S. S. Kanhere, M. A. Kaafar, E. Bertino, and S. Jha, "Gargoyle: A network-based insider attack resilient framework for organizations," in *IEEE LCN, 2018*.

[36] Nist - national vulnerability database.

[37] R. Cieslak, "Dynamic linker tricks: Using ld preload to cheat, inject features and investigate programs," *Retrieved March*, vol. 12, p. 2015, 2015.

[38] (2023) P4 16 Language Design. [Online]. Available: https://forum.p4.org/t/p4-16-language-design-why-tables-are-not-allowed-to-be-updated/-from-data-plane/783

[39] M. V. Dumitru, D. Dumitrescu, and C. Raiciu, "Can we exploit buggy p4 programs?" in *ACM SOSR, 2020*.

[40] T. Sasaki, C. Pappas, T. Lee, T. Hoefler, and A. Perrig, "Sdnsec: Forwarding accountability for the sdn data plane," in *IEEE ICCCN, 2016*.

[41] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "Sphinx: detecting security attacks in software-defined networks." in *NDSS, 2015*.

[42] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, and P. Athanas, "P4pktgen: Automated test case generation for p4 programs," in *ACM SOSR, 2018*.

[43] K. S. Kumar, R. K, P. S. Prashanth, M. T. Arashloo, V. U., and P. Tammana, "Dbval: Validating p4 data plane runtime behavior," in *ACM SOSR, 2021*.

[44] Aurora 610. [Online]. Available: https://netbergtw.com/products/aurora-610/

[45] T. Ball and J. R. Larus, "Efficient path profiling," in *IEEE MICRO, 1996*.