Computing (2021) 103:291–311
https://doi.org/10.1007/s00607-020-00835-4

SPECIAL ISSUE ARTICLE

Lightweight edge authentication for software defined networks

Amar Almaini[1,2] · Ahmed Al-Dubai[1] · Imed Romdhani[1] · Martin Schramm[2] · Ayoub Alsarhan[3]

Received: 14 February 2020 / Accepted: 10 July 2020 / Published online: 8 August 2020

Abstract
OpenFlow is considered as the most known protocol for Software Defined Networking (SDN). The main drawback of OpenFlow is the lack of support of new header definitions, which is required by network operators to apply new packet encapsulations. While SDN's logically centralized control plane could enhance network security by providing global visibility of the network state, it still has many side effects. The intelligent controllers that orchestrate the dumb switches are overloaded and become prone to failure. Delegating some level of control logic to the edge or, to be precise, the switches can offload the controllers from local state based decisions that do not require global network wide knowledge. Thus, this paper, to the best of our knowledge, is the first to propose the delegation of typical security functions from specialized middleboxes to the data plane. We leverage the opportunities offered by programming protocol-independent packet processors (P4) language to present two authentication techniques to assure that only legitimate nodes are able to access the network. The first technique is the port knocking and the second technique is the One-Time Password. Our experimental results indicate that our proposed techniques improve the network overall availability by offloading the controller as well as reducing the traffic in the network without noticeable negative impact on switches' performance.

Keywords Software-defined networking (SDN) · Data plane programmability · Port scan · Security · P4

Mathematics Subject Classification 68M10 · 68M14 · 68M25

1 Introduction

Software-Defined Networking (SDN) is an emerging network paradigm in which the control of the network is separated from the forwarding plane. In SDN, the logically

Extended author information available on the last page of the article

centralized controllers orchestrate the data plane and process the packet forwarding decision for the switches.

OpenFlow [1,2] is the main and most known interface to the switches in SDN. Openflow provides a uniform abstraction to configure different network devices. This means that the controller inserts and updates forwarding rules into flow tables independent of the switch vendor. The main idea behind OpenFlow is to give network administrators the possibility to remotely reconfigure the forwarding tables at runtime, collect network statistics, and when no match exists in the table for any packet, to redirect the packet to the controller for further processing. This reactive approach exhibits additional undesirable latency.

In OpenFlow any stateful processing is de facto delegated to the intelligent controller and the switches are just limited to installing forwarding rules provided by the controller. This centralized decision-making process is advantageous when controller's global network visibility is required and time is not critical. However, when the global network visibility is not needed or applications rely only on local flow/port states which need to react at the packet-level time scale, the centralized decision-making process might become a bottleneck [3]. As the latency caused by the reliance on an external controller eliminates the possibility to enforce software-implemented control plane tasks at wire-speed. For instance, a 64 bytes packet takes about 5 ns on a 100 Gbps speed, roughly the time needed for a signal to reach a control entity placed one meter away. In addition, the execution of an even though simple software-implemented control task may take way more time than this.

In the recent years, a lot of research investigated the possibility of delegating stateful processing to the switches, in order to reduce the switch-to-controller signaling overhead. One of the most valuable outcomes of this research effort is P4 [4–7]. This high-level language can be considered as a new phase in the development of SDN, where the software running over the switches becomes more flexible, extendable and powerful. P4 enables the SDN switches to perform advanced packet processing including stateful processing. To this end, P4 provides stateful memories (counters, meters, and registers), which persist the state across multiple packets of a flow.

Separating the control and forwarding planes with a centralized intelligence in the controller can offer wide opportunities to reduce network control and management complexity. However, in the context of security there is no need to centralize tasks that only need the local state to be accomplished and therefore there is no benefits from the network-wide knowledge that the controller provides [8–10]. On the contrary, this explicit involvement of the controller for each stateful processing and for any update of the match/action table leads to extra signaling overload and might cause the failure of the controller itself. Port knocking is an example of tasks that do not need the controller's network-wide knowledge.

Port knocking is an authentication technique used by network administrators [11–13]. It consists of a specified sequence of closed port connection attempts to specific IP addresses called a knock sequence. A host that wants to establish a connection delivers a sequence of packets addressed to an ordered list of prespecified closed ports. Once the exact sequence of packets is received, the firewall opens a specific port for the considered host. Before this event, all packets are dropped. One-time password is also a task that can be implemented without involving the controller.

As argued above, we believe that equipping the data plane with some intelligence can enhance the network overall performance, reduce the signaling load in the network [14], offload the controller and fulfill real-time requirements for some applications. Many applications that require sophisticated hardware can then run inexpensively and in a distributed manner on the data plane. In this paper, we propose to delegate the security tasks that usually need sophisticated equipment into the SDN switches by taking port knocking application as a practical example. For our use case, we have generalized the port knocking method to officiate like an authentication method for hosts or subnetworks that intend to establish a connection to a specific server. We show how a single SDN switch is programmed with P4 to efficiently perform the port knocking application and officiate like an authentication unit directly on the network ingress. To the best of our knowledge this is the first work, which delegates the authentication entirely to the data plane using P4.

2 Centralized vs. distributed approach

As we have mentioned before, the centralized decision-making provided by the controller in SDN is advantageous and can enhance the system security, when controller's global network visibility is required and time is not critical. However, the explicit involvement of the controller for any stateful processing is problematic. In the best case, this leads to extra signaling load and processing delay.

If we consider the scenario, in which our port knocking application is deployed directly on the logically centralized controller, we will see that the potentially large amount of signaling information (in principle up to all packets addressed to all n ports in the sequence) must be sent to the controller. Moreover, a timely instruction from the controller is needed for forwarding the packet after a correct knocking sequence, in order to avoid the drop of the first legitimate packet. Additionally, the controller must reserve memory resources to keep track on the state of all the nodes attempts to knock the secret sequence. Both, the extra signaling and the memory consumed are increasing with the length of the sequence and with the number of nodes attempts to initiate a connection. Furthermore, implementing this application in the controller brings no gain, as it does not benefit from global network visibility or high-level security policies [15], but uses just local states associated to specific flows on a single specific device.

Another argument for the delegation of some control to the data plane, when the applications rely only on local flow/port states is to be explained as follows. Even if stateful SDN moves some states down to the data plane, the core logic is maintained by the (logically centralized) control plane. Therefore, it is challenged by potential issues resulted by an inconsistent global view of the network, e.g., in a physically distributed controller setting due to a controller crash or disconnection [16].

Moving states to the data plane, however, may eliminate the vulnerability for very special applications; our port knocking application is an example. In this case, the operations performed by the switch are entirely local and independent from the global potentially inconsistent state maintained by the control plane.

3 Related work

The research community has extensively studied programmable data planes options in SDN. Recently there have been several approaches to overcome the limitations of OpenFlow and the disadvantages of involving the controller in all decisions and the associated extra latency. A great deal of emphasis on security functions in the data plane has been placed in the proposed works.

F. Paolucci et al. [17] proposed the adoption of P4 technology in an SDN multilayer packet-over-optical network to enable advanced data plane programmability. In particular, the work proposed an edge node architecture that includes a P4 switch with support of deep packet inspection. The P4-enabled node exploits direct stateful processing at wire speed without any intervention of the controller. Moreover, it proposed dynamic P4-based traffic engineering solutions for a multilayer scenario, such as traffic offloading. In addition, augmented firewalling capabilities are envisioned by using a P4 DDoS mitigation proof-of-concept to protect internal edge resources without the need for dedicated firewall hardware.

Experimental results showed impressive scalability performance with the size of the P4 program and in terms of switch latency to perform P4 operations, especially in the NetFPGA implementation. For example, only a 5 μs overall switch latency was experienced running the cybersecurity P4 code, with no performance degradation using even 10^4 flow entries.

Lin, Frank Po-Chen et al. [18] have discussed the problem of the computation-resource limited controller, which can be congested by heavy flows and then experiences serious delay. To enhance network scalability and reduce computation delay on SDN networks under Quality of Service (QoS) requirements, they have proposed a hierarchical edge-cloud SDN controller system design.

By sharing computational resources in the edge and the cloud, the system architecture provides a flexible mechanism for devices to allocate their computational tasks according to traffic loads. They have proved their system to be effective even when working on a large-scale SDN, without degrading the overall performance. Moreover, the system's performance remains highly stable even under highly fluctuating traffic loads.

Sivaraman et al. [8] proposed HashPipe, a heavy hitter detection algorithm using emerging programmable data planes with P4. They suggest running heavy-hitter monitoring at all switches in the network all the time, to respond quickly to short-term traffic variations. They argued that many network applications could benefit from finding the set of flows contributing significant amounts of traffic to a link.

In [19], the authors have proposed security functions like access control and port filtering in the data plane using P4. They described their approach as a second-generation firewall, which works as packet filter and has stateful memory.

An SDN network based system for the mitigation of denial of service attacks, specifically anti-spoofing techniques in the SDN data-plane, has been proposed in [20].

Li et al. [21] proposed LOSSRADAR, a lightweight packet loss detection service that quickly reports the locations and 5-tuple flow information of individual lost packets.

Bianchi et al. [3] proposed in their OpenState study the use of eXtended Finite State Machines (XFSM). They argue that programmers should be able to specify how states on the switch should be handled and this specification should be executed directly on the device without any controller interaction. They suggest modifications to OpenFlow 1.1 to allow stateful processing of flows in the switches.

Kabasele et al. [22] proposed a two-level network-based Intrusion Detection System (IDS) for Industrial Control Systems (ICS) based on SDN. The first level of the IDS is a whitelist-based filter implemented in P4 on network switches. If a switch cannot find a matching whitelist entry for a packet, it forwards the packet to the second level of IDS. This second level is a security engine running on a dedicated host. The security engine determines whether the packet is malicious or benign and instruct the controller to update the tables on the switches to accept or block connections.

In [23], the authors have proposed an approach to secure the access control system. Controller modules dedicated to authentication, registration and tracking of switches, hosts and users, as well as management of data flows were introduced. SDN capabilities were used to achieve a tight binding between actual machines and network device ports, combining with common RADIUS based user authentication, that enables network security. This approach is again centralized in the controller, which might become a bottleneck and prone to failure.

4 Specification using finite state machine

In this paper we have used the Finite State Machine (FSM) as model to specify the desired deterministic behavior of port knocking as an authentication unit in the switch. The FSM contains a finite number of states and it can be in exactly one state at any given time. In response to some external inputs, the FSM can change from one state to another (transition). The FSM is called deterministic, when at a state and upon an input, the machine follows a unique transition to a next state [24].

This abstract model fits exactly to the desired behavior that we want to achieve. Our motivation was to design this functionality and implement it in P4 to perform authentication in the switch without involving the controller. In order to make the design more reconfigurable, we have kept the port-sequence that must be knocked by the initiating node flexible in terms of length and order. The sequence can be adjusted at any time after the switch is deployed, in case the sequence was cracked by a malicious node. We have also made the following two elements configurable:

(1) The set of untrustworthy nodes (the set of nodes that need to knock the correct secret sequence of ports before they can initiate the connection to the server).
(2) The set of nodes that requires all the nodes that initiate connections to them to be authenticated first.

We have taken the FSM as an abstract model, which consists of number of states including default and final states. The transition between states is triggered by two types of events. The first event occurs when the connection initiator, which needs to authenticate being a member of the group of trustworthy nodes has knocked the correct port in the sequence. This will lead to a transition to the next state, as illustrated

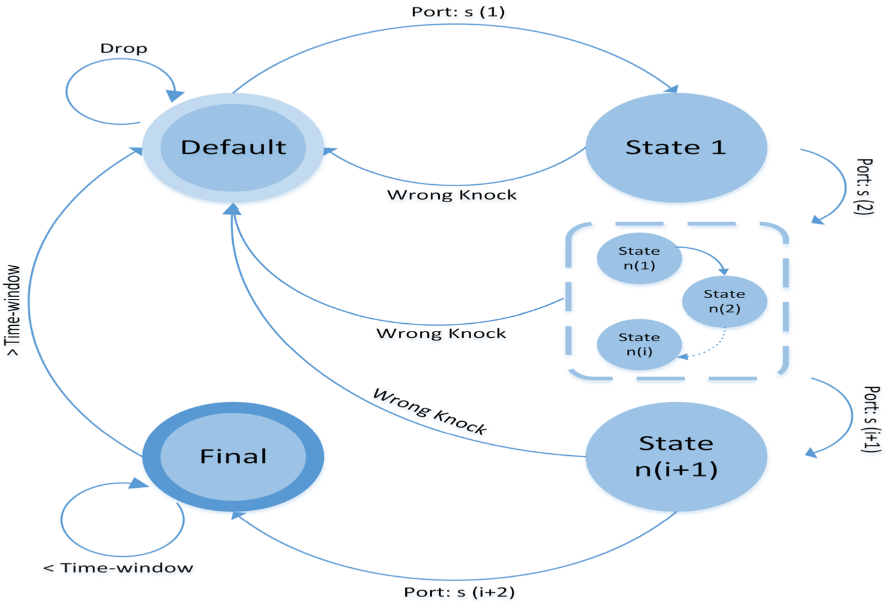


Fig. 1 FSM to describe the deterministic behavior of port knocking (color figure online)

in Fig. 1. A transition from default state to state 1 can take place, if and only if the connection initiator has knocked the first correct port in the sequence (the pointer from state default to state 1). Consequentially, the transition from state 1 to state 2 can take place, if and only if the connection initiator has knocked the second correct port in the sequence and so on.

The second event occurs when the initiator has knocked the wrong port in the sequence while it is in any state except the default and final states. Consequentially, a roll back action will be triggered and the state will set to default (Fig. 1, the pointers from each state to the default state). As long as the initiator is in the default state and has not knocked the first correct port in the sequence, no transitions occur. The final state can just be achieved after the correct sequence of ports has been knocked from the connection initiator as shown in Fig. 1. We will discuss this in more details later when we demonstrate the functionality of our proposed application.

5 The implementation

To demonstrate the ideas proposed in this paper, we have used the behavioral model (bmv2) and Mininet. We have simulated a simple network consisting of one switch and four hosts out of the larger topology illustrated in Fig. 2. We opted for this topology for two key reasons. First, we propose an authentication node that runs entirely on a single switch. Obviously, a larger topology would allow the demonstration of multiple authentication nodes running on different switches, each responsible for a different subnetwork. However, this would not contribute to the clarification of the concept. The

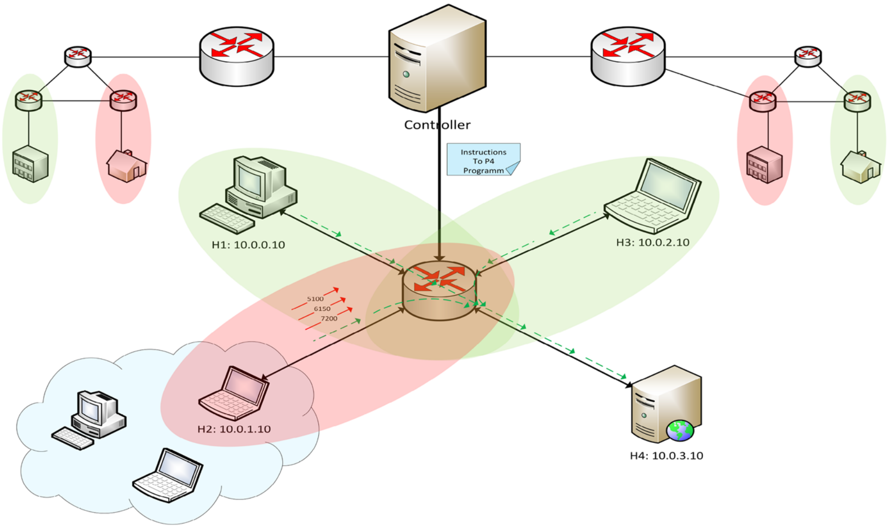


Fig. 2 Reference topology. Trustworthy nodes are highlighted with green oval shapes, while untrustworthy nodes are highlighted with red oval shapes (color figure online)

second reason is to demonstrate the case, if the switch fails. In this case, the controller in runtime can activate another switch, where the program was previously installed, so that it takes over the authentication task. This scenario demonstrates the network's resilience, which is beyond the scope of this work.

In our implementation[1], we have proposed a scenario in which each host except H2 can initiate connection to any other host, the switch is just forwarding the packets to the intended destination. In the illustrated scenario, we assumed that H2 is suspected to be an untrustworthy node that needs to authenticate itself first before it can initiate a connection to the server H4.

In the proposed authentication application, we have used the principle of tickets. This mean that only nodes that can present a valid ticket can initiate a connection to other nodes. We differentiate between benign nodes and those that are suspected to be untrustworthy nodes. Benign nodes are granted valid tickets initially. This demonstrates that they can initiate a connection to any other node without the need to authenticate themselves.

In contrast, nodes that are suspected to be untrustworthy (e.g. because of any abnormal activities in the past) are not granted valid tickets. This confirms that they need to authenticate themselves first via port knocking. Only if they were able to knock the correct secret sequence of ports, they will be granted with valid tickets to initiate the connection.

In our scenario, after H2 has finished the port knocking process successfully, it gains a valid ticket (Permission) to initiate a connection to H4 for a specific predefined period of time. After this time, the ticket is invalidated and H2 needs to authenticate

[1] https://github.com/amar-almaini/Delegation-of-Authentication-to-the-Data-Plane-in-SDN.

itself again. This measure avoids a scenario in which a tampered node can use an old ticket to initiate unpermitted connection to the server.

Inspired by the FSM We have implemented another security measure to avoid the case in which a node accidentally or intentionally knocks the correct sequence by knocking arbitrarily long sequence of ports that includes the secret correct sequence of ports. To specify this, we have defined the condition to trigger a transition to the next state as pair <Port, state>.
In our implementation, a knocked port is considered correct, if and only if it is the next expected correct port-number in the sequence and the assigned state in the P4 program to the knocking node is the correct state in the <Port, state> pair. This enforces that only nodes, which knock the port-sequence in the predefined order are successively authenticated otherwise the authentication fails and the node needs to start the port knocking from first port in the sequence.

As long as the correct port-sequence is not provided and the ticket is not granted, all packets sent by the node are dropped in the switch without sending any feedback to the node. Only after the correct port-sequence is sent and the ticket is granted the following packets are forwarded to the destination. This prevents the malicious node from making any conclusions about any correct parts from the sequence. To demonstrate this process, we suppose a secret port-sequence of (5100, 6150 and 7200). Initially each suspected node that need to authenticate itself and gain a ticket has the state 0 (Default) assigned. The switch expects initially the Pair <5100, 0>. When the node knocks first the port 5100, this will lead to the correct pair and a transition to state 1 is made. In the next step, the switch expects the pair <6150, 1>, therefore only if the node knocks port 6150, the condition is fulfilled and a transition to state 2 is made. Otherwise a rollback to the Pair <5100, 0> is triggered and the state 0 is assigned to the node as illustrated in Table 1.

Without the rollback transition, the switch will keep the state 1 assigned to that node also when it knocks wrong ports after it has knocked the port 5100. This way, if the node knocks anytime accidentally the port 6150, this will lead mistakenly to a transition to state 2, which makes it possible to knock the correct sequence by knocking long sequence of ports, that could accidentally include the correct sequence in the proper order.

On the contrary, implementing this functionality on the logically centralized controller will lead to a new packet-in message for each new packet arriving to the switch, as the switch up to this time has no rules installed on how to deal with this packet. Roughly, this will produce up to ℓ (the length of the port-sequence) more messages towards the controller, and similarly, for each node that attempts to initiate a connection to the server. Additionally, the controller needs to save the state for each port knocking operation, to finally install a new rule in the switch, which forwards or blocks the packets from this node. An attacker can use such an implementation to perform a DoS attack by consuming computation resources on the controller using a set of spoofed IPs.

Table 1 Port knocking. Knocking the wrong port 6200 leads to Rollback to state 0

Knocked port	Assigned state	Expected pair	Resulting pair	Resulting state	Action	Ticket granted
5100	0	$< 5100, 0 >$	$< 5100, 0 >$	1	Drop	False
6150	1	$< 6150, 1 >$	$< 6150, 1 >$	2	Drop	False
6200	2	$< 7200, 2 >$	$< 6200, 2 >$	0	Drop	False
7200	0	$< 5100, 0 >$	$< 7200, 0 >$	–	Drop	False
*	0	$< 5100, 0 >$	$< *, 0 >$	–	Drop	False
5100	0	$< 5100, 0 >$	$< 5100, 0 >$	1	Drop	False
6150	1	$< 6150, 1 >$	$< 6150, 1 >$	2	Drop	False
7200	2	$< 7200, 2 >$	$< 7200, 2 >$	3	Drop	True
Any	3	–	–	–	Forward	True

(∗) All ports except the correct port 5100

5.1 Dynamically reconfiguring the switch

Implementing the authentication application with P4 in the switch obviously has numerous advantages such as reducing the malicious traffic in the network as all traffic produced from suspicious nodes is filtered out at the network ingress without the need to forward this traffic inside the network to specialized middleware where it can be filtered out.

After the controller has added the authentication rules into the switch in the responsible table, it is not involved anymore and the switch doesn't need to ask the controller for each successor incoming packet to make a decision or add a new forwarding rule. This will reduce the load on the controller. Moreover, in case the switch goes down for any reason, the authentication application can be activated at runtime in any other switch in the network without influencing the network availability.

Despite all the advantages, it is worth indicating that this method of authentication does not provide adequate protection. To demonstrate this, we provide an attack example, exploiting the vulnerability of the authentication function in the switch to the memory saturation attack. We consider a scenario of an attacker who knows the correct sequence of ports to knock. This information could be gained through several ways, such as sniffing the traffic.

Assume the attacker sends a large amount of connection attempts (i.e., packets) to the first port in the predefined sequence (i.e., 5100) from a set of spoofed IPs, towards the switch. Upon receiving a packet from a source IP, the switch checks its state table to retrieve the state of the incoming flow. If there is no record for this IP address, the switch assigns the next state to the corresponding IP. Now, since all the incoming packets are destined to the correct port, the state of all the incoming packets (i.e., the corresponding IPs) will be updated to the next state (i.e., State 1). Therefore, we will have a large number of entries inside the state table of the switch. This way, a conscious attacker is able to force the generation of thousands of records in the state table, and thus exhausting the switch memory [25].

The consequence of this vulnerability can be reduced by configuring the switch with a timeout, if the switch does not receive the second packet from the same flow knocking the next port, the switch will reset the state of the flow and remove flow's state from the state table.

Additionally, distributing the authentication functionality over more than one switch in the network, with efficient load balancing between the switches can also reduce the effect of this attack significantly.

To make our implementation more realistic and scalable and in order to be able to handle the continuous changes occurring in the network, we kept it dynamic and reconfigurable at runtime. Nodes that are suspected as untrustworthy and need to authenticate themselves before initiating a connection are defined as pairs of source and destination, with IPv4-Addresses. The source represents the suspected node or subnetwork whereas the destination represents the server or the protected node.

This means that one node can be suspected in the context of connection initiation to a specific node/server, but the same node may not be suspected in the context of connection initiation to another node/server. To add one node (connection initiation)

```
apply(ticket_deny_table) {
        hit {
                host_need_processing();
        }

        miss {
                apply(grant_ticket_for_not_suspicious_table);
        }
```

Fig. 3 Hit/Miss construct. Hit identifies the untrustworthy nodes, while Miss identifies the trustworthy nodes

```
table_add update_state_table update_state 5100 0 => 1 0
table_add update_state_table update_state 6150 1 => 2 0
table_add update_state_table update_state 7200 2 => 3 1
```

Fig. 4 CLI commands to set the initial knock-sequence

to the list of suspected nodes, it's sufficient that the controller adds one rule to the responsible table and in contrast, it's sufficient to remove this rule from the table to remove the node from the list. This can be done dynamically at runtime to reconfigure the switch and handle new incidents occurring in the network.

To implement this behavior, we have used the Hit/Miss construct in P4 [2]. When there is a record in the table for a specific node (Hit), it will be handled as suspected and it must authenticate itself first to gain a ticket. In contrast, when there is no record in the table for this node (Miss), it will be handled as a benign node that has a ticket by default (see Fig. 3).

Using the same approach, the secret sequence of ports that have to be knocked in specific order to gain a valid ticket can be also configured by the controller at runtime. The controller can delete the current sequence or a subsequence at any time and set a new sequence with the same length or a longer sequence to increase the security and difficulty of cracking the sequence, i.e., an oblivious attacker will need to guess the correct sequence out of 65535^{ℓ} (ℓ is the length of the sequence) different possible permutations.

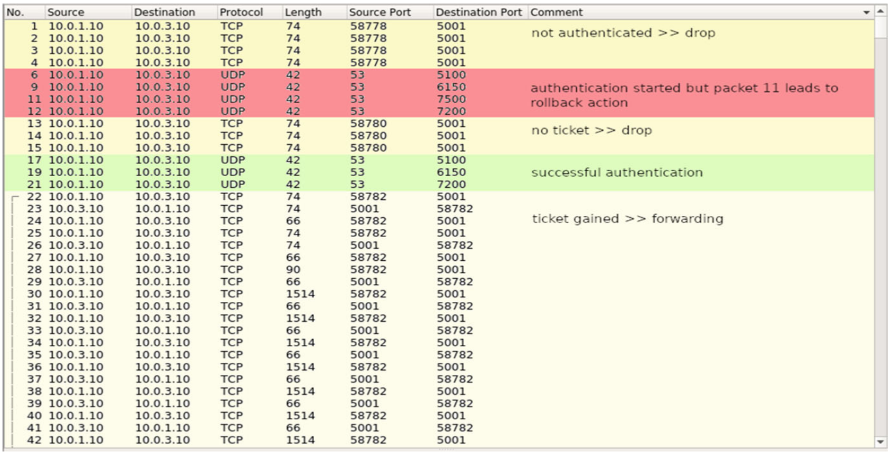
To demonstrate this, we use the runtime command line interface (CLI). In the initial configuration, we have set the sequence (5100, 6150, and 7200) with the following commands (see Fig. 4).
Where 5100 is the first port in the sequence, 0 is the initial state, 1 is the state after the transition and the last 0 in the first line is the ticket validity (0 = not valid, 1 = valid). To change this sequence, the controller need to delete these records and insert new rules with the new sequence as shown in Fig. 5
This will cause a change in the sequence at runtime and only the suspected nodes, that are able to knock the new sequence in the proper order will gain a valid ticket.

In our initial configuration, we have defined the node 10.0.1.10 (H2) in the context of connection attempt to 10.0.3.10 (H4) as suspected, then we sent a TCP stream using

```
table_delete update_state_table 0
table_delete update_state_table 1
table_delete update_state_table 2

table_add update_state_table update_state 6100 0 => 1 0
table_add update_state_table update_state 8150 1 => 2 0
table_add update_state_table update_state 9500 2 => 3 0
table_add update_state_table update_state 5100 3 => 4 1
```

Fig. 5 CLI commands to change the knock-sequence



No.	Source	Destination	Protocol	Length	Source Port	Destination Port	Comment
1	10.0.1.10	10.0.3.10	TCP	74	58778	5001	not authenticated >> drop
2	10.0.1.10	10.0.3.10	TCP	74	58778	5001	
3	10.0.1.10	10.0.3.10	TCP	74	58778	5001	
4	10.0.1.10	10.0.3.10	TCP	74	58778	5001	
6	10.0.1.10	10.0.3.10	UDP	42	53	5100	authentication started but packet 11 leads to rollback action
9	10.0.1.10	10.0.3.10	UDP	42	53	6150	
11	10.0.1.10	10.0.3.10	UDP	42	53	7500	
12	10.0.1.10	10.0.3.10	UDP	42	53	7200	
13	10.0.1.10	10.0.3.10	TCP	74	58780	5001	no ticket >> drop
14	10.0.1.10	10.0.3.10	TCP	74	58780	5001	
15	10.0.1.10	10.0.3.10	TCP	74	58780	5001	
17	10.0.1.10	10.0.3.10	UDP	42	53	5100	successful authentication
19	10.0.1.10	10.0.3.10	UDP	42	53	6150	
21	10.0.1.10	10.0.3.10	UDP	42	53	7200	
22	10.0.1.10	10.0.3.10	TCP	74	58782	5001	ticket gained >> forwarding
23	10.0.3.10	10.0.1.10	TCP	74	5001	58782	
24	10.0.1.10	10.0.3.10	TCP	66	58782	5001	
25	10.0.1.10	10.0.3.10	TCP	74	58782	5001	
26	10.0.3.10	10.0.1.10	TCP	74	5001	58782	
27	10.0.1.10	10.0.3.10	TCP	66	58782	5001	
28	10.0.1.10	10.0.3.10	TCP	90	58782	5001	
29	10.0.3.10	10.0.1.10	TCP	66	5001	58782	
30	10.0.1.10	10.0.3.10	TCP	1514	58782	5001	
31	10.0.3.10	10.0.1.10	TCP	66	5001	58782	
32	10.0.1.10	10.0.3.10	TCP	1514	58782	5001	
33	10.0.3.10	10.0.1.10	TCP	66	5001	58782	
34	10.0.1.10	10.0.3.10	TCP	1514	58782	5001	
35	10.0.3.10	10.0.1.10	TCP	66	5001	58782	
36	10.0.1.10	10.0.3.10	TCP	1514	58782	5001	
37	10.0.3.10	10.0.1.10	TCP	66	5001	58782	
38	10.0.1.10	10.0.3.10	TCP	1514	58782	5001	
39	10.0.3.10	10.0.1.10	TCP	66	5001	58782	
40	10.0.1.10	10.0.3.10	TCP	1514	58782	5001	
41	10.0.3.10	10.0.1.10	TCP	66	5001	58782	
42	10.0.1.10	10.0.3.10	TCP	1514	58782	5001	

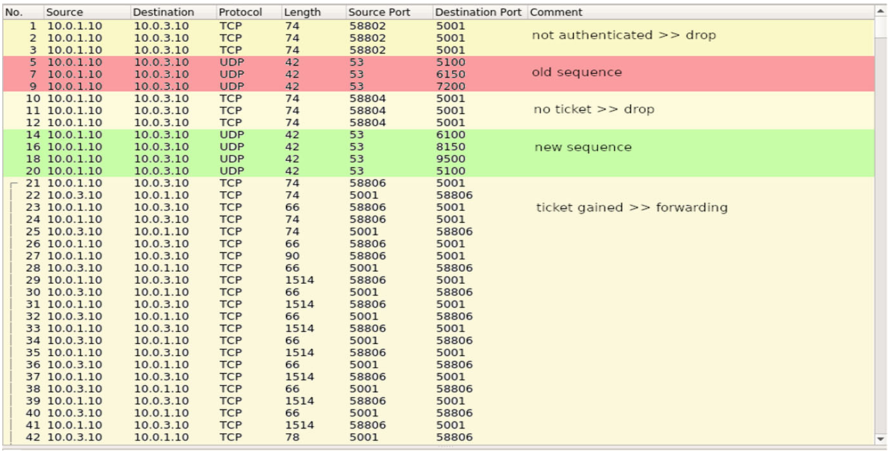
Fig. 6 Wireshark screenshot shows how H2 knocks the port-sequence to gain a valid ticket

iperf from H2 towards H4. Figure 6 illustrates how all packets are dropped, initially, because up to this point of time H2 have not yet authenticated itself.

Using a second terminal, we sent UDP packets from H2 to H4 with destination ports 5100, 6150 and 7200 to gain a valid ticket. The first attempt went wrong because another packet with wrong port arrived in between. This triggered a rollback action forcing the host to initiate the sequence again from the beginning. Only when the sequence was knocked in the proper order, the host was authenticated and the ticket is supplied. All the successor packets were forwarded normally to the destination.

The ticket is valid just for a predefined time. Consequently, the switch checks the arriving time for all successor packets and as soon as the validity time is expired, the ticket is invalidated and the packets are dropped until the node gains a new valid ticket.

While the switch was running, we have reconfigured the switch with a new secret sequence as shown in Fig. 5, which is normally done by the controller as a reaction to specific incidents or anomalies in the network. Consequently, any suspected node that attempts to initiate a connection must first supply the new sequence to gain a valid ticket as shown in Fig. 7, a previously cracked sequence is not applicable anymore.



No.	Source	Destination	Protocol	Length	Source Port	Destination Port	Comment
1	10.0.1.10	10.0.3.10	TCP	74	58802	5001	not authenticated >> drop
2	10.0.1.10	10.0.3.10	TCP	74	58802	5001	
3	10.0.1.10	10.0.3.10	TCP	74	58802	5001	
5	10.0.1.10	10.0.3.10	UDP	42	53	5100	old sequence
7	10.0.1.10	10.0.3.10	UDP	42	53	6150	
9	10.0.1.10	10.0.3.10	UDP	42	53	7200	
10	10.0.1.10	10.0.3.10	TCP	74	58804	5001	no ticket >> drop
11	10.0.1.10	10.0.3.10	TCP	74	58804	5001	
12	10.0.1.10	10.0.3.10	TCP	74	58804	5001	
14	10.0.1.10	10.0.3.10	UDP	42	53	6100	new sequence
16	10.0.1.10	10.0.3.10	UDP	42	53	8150	
18	10.0.1.10	10.0.3.10	UDP	42	53	9500	
20	10.0.1.10	10.0.3.10	UDP	42	53	5100	
21	10.0.1.10	10.0.3.10	TCP	74	58806	5001	ticket gained >> forwarding
22	10.0.3.10	10.0.1.10	TCP	74	5001	58806	
23	10.0.1.10	10.0.3.10	TCP	66	58806	5001	
24	10.0.1.10	10.0.3.10	TCP	74	58806	5001	
25	10.0.3.10	10.0.1.10	TCP	74	5001	58806	
26	10.0.1.10	10.0.3.10	TCP	66	58806	5001	
27	10.0.1.10	10.0.3.10	TCP	90	58806	5001	
28	10.0.3.10	10.0.1.10	TCP	66	5001	58806	
29	10.0.1.10	10.0.3.10	TCP	1514	58806	5001	
30	10.0.3.10	10.0.1.10	TCP	66	5001	58806	
31	10.0.1.10	10.0.3.10	TCP	1514	58806	5001	
32	10.0.3.10	10.0.1.10	TCP	66	5001	58806	
33	10.0.1.10	10.0.3.10	TCP	1514	58806	5001	
34	10.0.3.10	10.0.1.10	TCP	66	5001	58806	
35	10.0.1.10	10.0.3.10	TCP	1514	58806	5001	
36	10.0.3.10	10.0.1.10	TCP	66	5001	58806	
37	10.0.1.10	10.0.3.10	TCP	1514	58806	5001	
38	10.0.3.10	10.0.1.10	TCP	66	5001	58806	
39	10.0.1.10	10.0.3.10	TCP	1514	58806	5001	
40	10.0.3.10	10.0.1.10	TCP	66	5001	58806	
41	10.0.1.10	10.0.3.10	TCP	1514	58806	5001	
42	10.0.3.10	10.0.1.10	TCP	78	5001	58806	

Fig. 7 Wireshark screenshot shows how H2 knocks the new port-sequence to gain a valid ticket

5.2 Port scan mitigation through authentication

Attackers routinely perform random “portscans” of IP addresses to find vulnerable servers to compromise. Portscan [25,26] work by sending requests to a big range of port addresses on a host, analyzing the responses to identify the open ports on this node. Our proposed application is able to mitigate such attacks conducted from untrustworthy nodes.

Non-suspected nodes are able to initiate a connection to host/server without authentication. However, suspected nodes in the context of connection initiation to a specific node/server are initially blocked and all incoming packets from these nodes are dropped by the switch before reaching the node. As long as a suspected node have not yet authenticated itself, the protected node for it is invisible.

To demonstrate this, we have used nmap from H2 to perform a portscan on the protected host H4 (initial configuration). Because to this point of time H2 was not authenticated, all packets sent from nmap were dropped on the switch and the scanner returned the message “Host seems down”.

6 Evaluation

The setup we have used for this evaluation is Mininet with bmv2 (behavioral model) installed as virtual machine in VirtualBox. The host machine has an Intel i7 CPU and 8 GB RAM.

We will use two methods to evaluate the performance of the proposed work. The first one is comparing the performance of the switch between forwarding packets from a node initially defined as benign, which does not need to knock the secret sequence of ports and forwarding packets from a node initially defined as untrustworthy, which need to knock the secret sequence of ports and gain a ticket first.

As long as the ticket is valid, the only check made in the switch for all successor packets is comparing the packet's arriving time with the ticket's expiring time. The purpose of this experiment is to investigate, if the check of ticket validity negatively impacts the performance of the switch.

The second method is to compare the performance between a switch that implements our proposed implementation and the same switch performing only packet forwarding without any additional functionality.

6.1 Performance comparison of the switch with and without check, if time window is valid

To compare the switch performance in both cases, we have measured the throughput by sending fifty TCP streams for ten seconds duration per stream from the benign host H3 to H4, and fifty TCP streams from the untrustworthy host H2 to H4 (see the topology in Fig. 2), after the ticket was gained. After that, we have measured the average of all throughputs. As we can see in the Fig. 10a, b, the throughput average is approximately the same. This means that after a suspicious node was successfully authenticated, there is no negative impact on the performance of forwarding functionality in the switch.

In Fig. 8, we can see the performance of the switch when both nodes H2 and H3 send TCP streams, almost in parallel to H4. H3 starts sending while H2 is not authenticated, yet. We perform the authentication (knocking the port-sequence in the proper order) using a Python script from a second terminal to gain a valid ticket for H2. After that, we switch back to the first terminal to start the stream from H2 to H4. Switching between the two terminals manually requires a few seconds. This explains the delayed start of the stream sent from H2 as shown in Fig. 8.

Subsequently, we resend the streams this time for a duration of two minutes and we perform the authentication for H2 in advance. The intention for this experiment was to observe the impact of expiring the ticket while the stream is passing through the switch. Figure 9 shows how the ticket expired two times within the two minutes (validity time 50 s,). This has a noticeable negative impact on the overall throughput, which is the expected behavior in this case.

As we mentioned before we perform the authentication manually with a Python script, which requires additional time and scales up the effect. However, this effect can be reduced remarkably by increasing the validity time of the ticket.

6.2 Performance comparison of the switch with and without the authentication functionality

Our motivation for this evaluation was to find out if our application has any negative impact on the performance of the switch, and if yes, how large? For this, we have replaced the switch implementing our application with the same switch (bmv2), but without any additional functionality besides the basic forwarding functionality.

Again, we have sent fifty TCP streams for ten seconds duration per stream, from the host H2 to H4. After that, we have measured the average of all throughputs. As

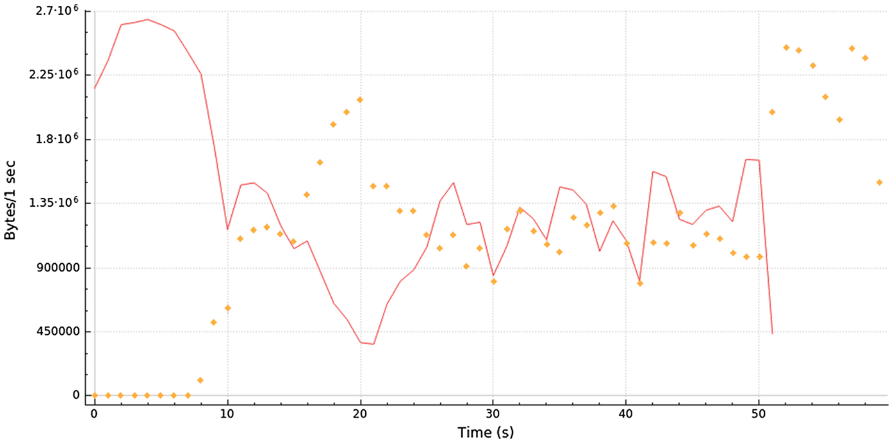


Fig. 8 Throughput of H2 and H3 as they send TCP streams to H4 (H2 dotted curve, H3 solid curve). In the first few seconds, H2 still not yet authenticated

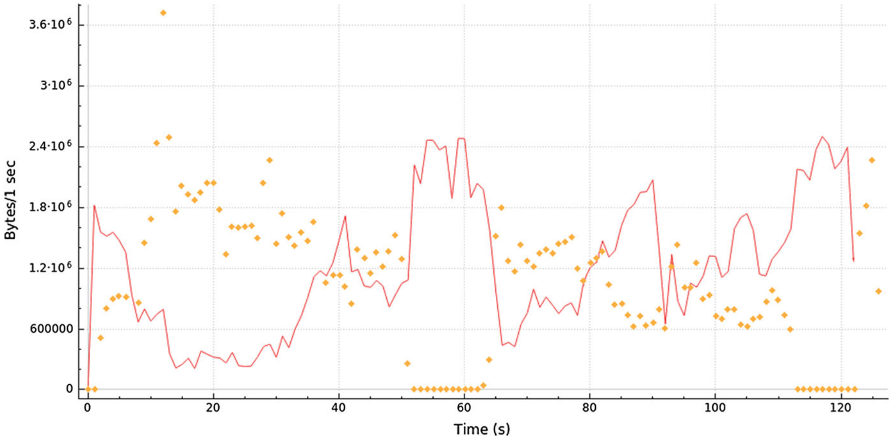


Fig. 9 Two times ticket expiration within two minutes (validity time 50 s)

we can see in the Fig. 10c, the throughput average is ≈ 20% better than the one we got with the switch implementing our application.

As discussed previously, despite the apparently lower throughput, the implementation of the authentication functionality in the switch still offers many advantages, compared with the implementation of the authentication functionality in the logically centralized controller.

To clarify, if the basic switch performs better in the case when many nodes are sending simultaneously to the server, we simultaneously started two streams from H2 and H3 to H4 with a duration of two minutes each. The result shows again that the behavior of the switch is not significantly negatively impacted in terms of throughput when our implementation is running as shown in Fig. 11.

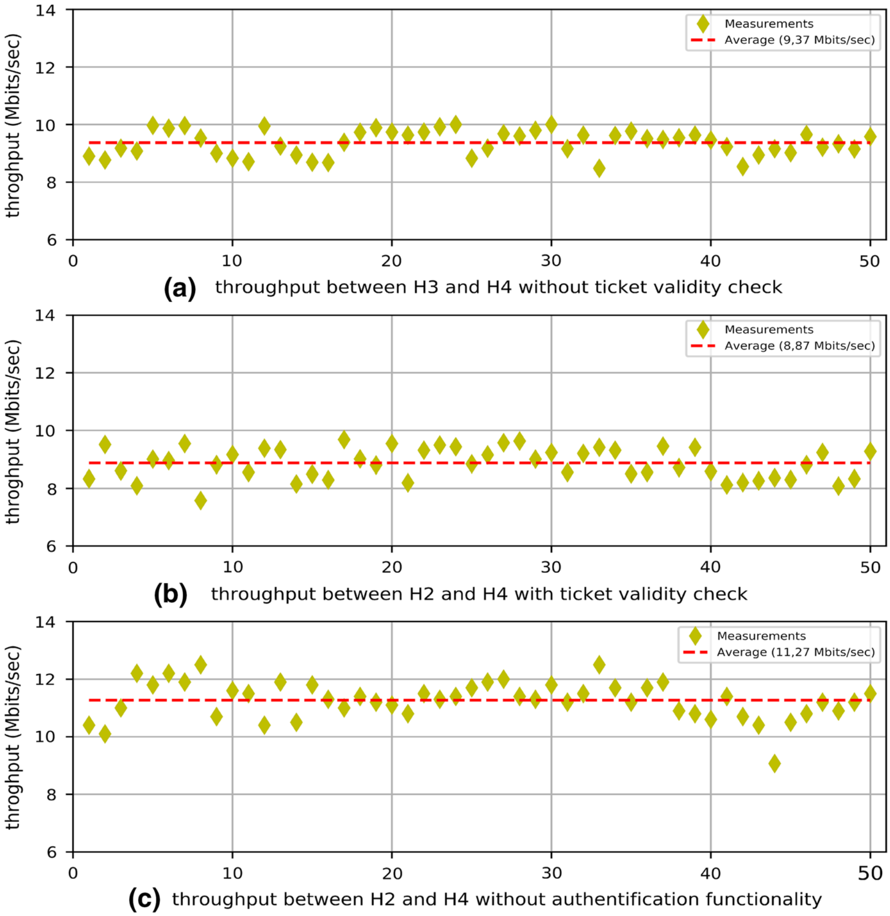


Fig. 10 Average throughput for fifty measurements in different scenarios

7 One-time password by the switch

One-Time Password (OTP) is an authentication/login-system, in which a password is only valid for one login or transaction. The Challenge with this approach is to track the next correct password for the switch as well as for the client. There are two approaches to solve this problem, the first solution is to use a password generator, which generates and distributes the password for the next login on request, the second approach is by maintaining a password list on the server as well as on the client. The last one is commonly known from online banking as TAN-Lists.

We propose to use this approach as an authentication method between the switch on the one side and the client, which intend to access the network on the other side. Our system uses cryptographic hashing functions (e.g. SHA-2, SHA-3 or BLAKE2) and is based on the Leslie Lamport algorithm [27].

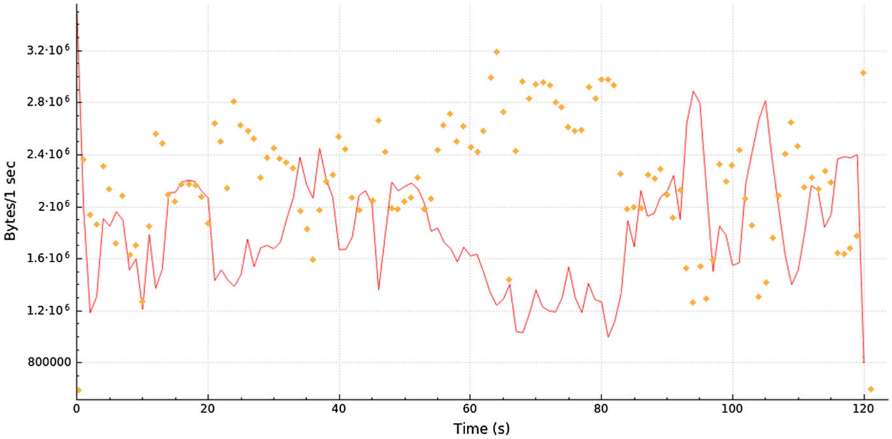


Fig. 11 Throughput of H2 and H3, as they send TCP streams to H4 (H2 dotted curve, H3 solid curve) without the authentication functionality deployed on the switch

The idea is to use a sequence of passwords $x_1, x_2, \ldots.., x_{100}$, where x_i is the password with which the client identifies himself for the ith time (100 is quite arbitrary). Our solution is to let the ith password x_i equal $F^{100-i}(x)$, where F is a cryptographic hashing function and F^n denotes n successive applications of F. Thus, the sequence of 100 passwords is $F^{99}(x), \ldots, F(F(F(x))), F(F(x)), F(x), x$, which is called hash chains. With this background, we design the algorithm to achieve the One-Time Password authentication in the data plane. The algorithm consists of several steps as shown in Fig. 12:

1. Generate a random initial value (seed) s.
2. A cryptographic hash function f will be repeatedly applied for n times to acquire the hash chain, e.g. $f^{100}(s)$, ..., $f(f(f(s)))$, $f(f(s))$, $f(s)$, s. The first $n-1$ values from the chain will be stored on the client side in a table.
3. The value of $f^{100}(s)$ will be stored on the switch as the Target value T.
4. For each client that intends to access the network, it needs first to authenticate itself by the switch. For this, the client presents for the first authentication the first password $f^{99}(s)$.
5. On the switch side, the switch compares the password sent from the client with its target value. For the first authentication round, this will mean comparing the following values $f(f^{99}(s)) = f^{100}(s)$. If the authentication was successful, the switch stores $f^{99}(s)$ instead of $f^{100}(s)$ as its new target.
6. For the next authentication to be successful, the client needs to present $f^{98}(s)$, then $f^{97}(s)$ and so forth. A successful authentication can be established for $n-1$ times, where n is the length of the hashing chain until the chain gets exhausted and a new chain must be generated.

By using a cryptographic hash function, this approach has inherently the property of not invertible, which means an eavesdropper has no chance to recalculate the chain, if he has successfully eavesdropped one password. Furthermore, the system is resistant against Replay-Attacks, because each password is only valid for one time.

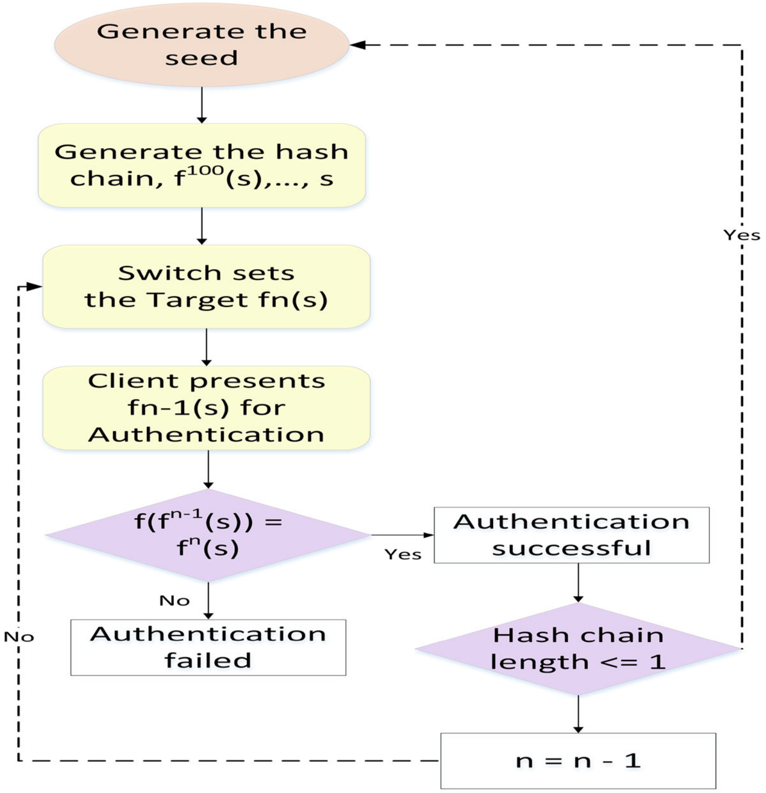

Generate the seed
Generate the hash chain, $f^{100}(s)$,..., s
Switch sets the Target fn(s)
Client presents fn-1(s) for Authentication
$f(f^{n-1}(s)) = f^{n}(s)$
Yes
Authentication successful
No
Authentication failed
Hash chain length <= 1
Yes
No
n = n - 1

Fig. 12 Flow design of the One-Time-Password authentication

To the other end, the process is vulnerable to man-in-the-middle attacks, when an attacker hooks up between the client and the switch and pretends to be the legitimate switch. Another Problem is that the client must know which password is the password in use. This might lead to problems, if there is a package loss. This problem can be solved by sending an acknowledgment back for each received password.

8 Comparison with existing work

In this section, we compare our proposed work with existing work that focuses on authentication using port knocking. In [28], the author describes a similar scheme where a client opens a port by attempting connections to a secret sequence of port numbers. The port-sequence is used as a password. The connection attempts are sent to the firewall. All connections will be ignored but logged, the firewall will not send any error packets to the client. Using another service that monitors the firewall log, the port-sequence can be extracted and associated with the incoming IP address. As soon as the client sends the correct sequence, this specific port-sequence acts as a trigger to manipulate the firewall rules and the client is authenticated.

This concept has some similarity to our work; however, the crucial difference is the need for a sophisticated firewall unit to perform the authentication. This adds additional cost and more complexity to the system. The other drawback is that when the firewall fails, the network becomes unreachable, as the clients cannot be authenticated. While it is certainly possible to keep a second firewall node ready in the network that can replace the failed firewall, this will add even more cost and complexity. On the contrary, our work does not cause any extra cost or complexity in the system.

To highlight further the advantages of our work in more details, we will consider two key network characteristics, namely the scalability and the reconfigurability. Scalability is very important aspect of any dynamic modern network. However, the firewall approach in [29] supplies merely limited support for the scalability. As whenever the network grows, new firewall nodes are necessary to serve the new network segments, which increase the costs and complexity of the network. On the contrary, scalability is an inherent characteristic of our approach, as whenever the network grows, the new segments will require new basic network elements such as switches, where our application can be deployed to serve the new segments.

Reconfigurability is another key characteristic especially in highly dynamic networks. The continuously changed network leads to fast adjustments in the assigned rolls and access rights in the network. However, the firewall approach supplies merely limited support for the reconfigurability, as each firewall needs to be reconfigured individually, normally by the network administrator (e.g. new port-sequence). On the contrary, in our approach, all switches can be centrally and individually reconfigured by the controller depending on the network current status.

9 Conclusion and future work

We have presented a technique with P4 for performing authentication and port scan mitigation inside the switch on the transport Layer of the TCP/IP protocol stack. We showed the flexibility and dynamic nature of our technique in runtime. Experimental evaluation showed that our technique has no noticeable negative impact on the performance of the switch, which makes it applicable to real world scenarios.

This paper shows the advantages of delegating some of the controller intelligence into the data plane, which can enhance the network overall performance, reduce the signaling load in the network, offload the controller, and fulfill real-time requirements. Some aspects, such as the secure distribution of the knock-sequences to the trustworthy nodes, and the timeout to trigger a rollback action when the time between knocking the ports exceeds a specific threshold, have been not considered in this work. However, these aspects are subjects for future work.

In addition, we plan to test more security functions in the data plane using P4, and to test the applicability of dynamically distributing different security functions on different switches in the network considering the incidents and anomalies occurring in the network.

References

1. von Tüllenburg F, Pfeiffenberger T (2017) Concepts for reliable communication in a software-defined network architecture. In: International conference on computer safety, reliability, and security. Springer, Cham
2. Nick McKeown et al (2008) OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Comput Commun Rev 38(2):69–74. https://doi.org/10.1145/1355734.1355746
3. Bianchi Giuseppe et al (2014) OpenState: programming platform-independent stateful openflow applications inside the switch. ACM SIGCOMM Comput Commun Rev 44(2):44–51
4. Bosshart Pat et al (2014) P4: Programming protocol-independent packet processors. ACM SIGCOMM Comput Commun Rev 44(3):87–95. https://doi.org/10.1145/2656877.2656890
5. P416 Language specification. Last accessed 15 April 2019. 2019. URL: https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.pdf
6. Hyun J, Hong JWK (2017) Knowledge-defined networking using in-band network telemetry. In: Network operations and management symposium (APNOMS), 2017 19th Asia-Pacific. IEEE
7. Baktir AC, Ozgovde A, Ersoy C (2018) Implementing service-centric model with P4: a fully-programmable approach. In: NOMS 2018-2018 IEEE/IFIP network operations and management symposium. IEEE
8. Sivaraman V et al. (2017) Heavy-hitter detection entirely in the data plane. In: Proceedings of the symposium on SDN research. ACM
9. Paolucci F, Cugini F, Castoldi P (2018) P4-based multi-layer traffic engineering encompassing cyber security. In: Optical fiber communication conference. Optical society of America
10. Yehuda Afek et al (2018) Detecting heavy flows in the SDN match and action model. Comput Netw 136:1–12. https://doi.org/10.1016/j.comnet.2018.02.018
11. Ali FHM, Yunos R, Alias MAM (2012) Simple port knocking method: against TCP replay attack and port scanning. In: Cyber security, cyber warfare and digital forensic (CyberSec) 2012 international conference on. IEEE
12. Aycock J, Jacobson M (2005) Improved port knocking with strong authentication. In: 21st Annual computer security applications conference (ACSAC'05). IEEE
13. Nabi-Abdolyousefi Marzieh, Mesbahi Mehran (2012) Network identification via node knockout. IEEE Trans Autom Control 57(12):3214–3219
14. Sivaraman A et al. (2015) Dc. p4: Programming the forwarding plane of a data-center switch. In: Proceedings of the 1st ACM SIGCOMM symposium on software defined networking research. ACM
15. Nayak AK, Reimers A, Feamster N, Clark R (2009) Resonance: dynamic access control for enterprise networks. In: 1st ACM workshop on research on enterprise networking (WREN09)
16. Levin D, Wundsam A, Heller B, Handigol N, Feldmann A (2012) Logically centralized? state distribution trade-offs in software defined networks. In: Proc. 1st workshop hot topics softw. defined netw., Helsinki, Finland, pp 1–6
17. Paolucci F et al (2019) P4 Edge node enabling stateful traffic engineering and cyber security. J Opt Commun Netw 11(1):A84–A95
18. Lin FPC, Tsai Z (2019) Hierarchical edge-cloud SDN controller system with optimal adaptive resource allocation for load-balancing. IEEE Syst J 14:265
19. Vörös P, Kiss A (2016) Security middleware programming using P4. In: International conference on human aspects of information security, privacy, and trust. Springer, Cham
20. Afek Y, Bremler-Barr A, Shafir L (2017) Network anti-spoofing with SDN data plane. In: INFOCOM 2017-IEEE Conference on computer communications, IEEE. IEEE

21. Li Y et al (2016) Lossradar: fast detection of lost packets in data center networks. In: Proceedings of the 12th international on conference on emerging networking experiments and technologies. ACM
22. Kabasele NG, Sadre R (2018) A two-level intrusion detection system for industrial control system networks using P4. In: ICS-CSR 2018
23. Kuliesius F, Dangovas V (2016) SDN enhanced campus network authentication and access control system. In: 2016 Eighth international conference on ubiquitous and future networks (ICUFN). IEEE
24. Lee David, Yannakakis Mihalis (1996) Principles and methods of testing finite state machines-a survey. Proc IEEE 84(8):1090–1123
25. Dargahi Tooska et al (2017) A survey on the security of stateful SDN data planes. IEEE Commun Surv Tutor 19(3):1701–1725
26. LACORE, UCV. A review of port scanning techniques
27. Lamport Leslie (1981) Password authentication with insecure communication. Commun ACM 24(11):770–772
28. Krzywinski M (2003) Port knocking: network authentication across closed ports. SysAdmin Mag 12(6):12–17
29. Khan Suleman et al (2017) Towards port-knocking authentication methods for mobile cloud computing. J Netw Comput Appl 97:66–78. https://doi.org/10.1016/j.jnca.2017.08.018

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations

Amar Almaini[1,2] · **Ahmed Al-Dubai**[1] · **Imed Romdhani**[1] · **Martin Schramm**[2] · **Ayoub Alsarhan**[3]

✉ Amar Almaini
A.Almaini2@napier.ac.uk; amar.almaini@th-deg.de

Ahmed Al-Dubai
A.Al-Dubai@napier.ac.uk

Imed Romdhani
I.Romdhani@napier.ac.uk

Martin Schramm
martin.schramm@th-deg.de

Ayoub Alsarhan
ayoubm@hu.edu.jo

[1] Edinburgh Napier University, Edinburgh, United Kingdom

[2] Deggendorf Institute of Technology, Deggendorf, Germany

[3] Hashemite University, Zarqa, Jordan