©2022 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Preventing Control Plane Overload in SDN Networks with Programmable Data Planes

Cristian Bermudez Serna
Chair of Communication Networks (LKN)
Technical University of Munich (TUM)
Munich, Germany
cristian.bermudez-serna@tum.de

Carmen Mas-Machuca
Chair of Communication Networks (LKN)
Technical University of Munich (TUM)
Munich, Germany
cmas@tum.de

***Abstract*—Software-Defined Networking (SDN) has redefined the architectural blueprint for designing networks suitable for future applications. Today, the idea of a centralized control plane managing its underlying resources is common for architectures in mobile and industrial networks. Guaranteeing resources availability for optimal operation of the control plane is of vital importance in SDN, since compromising the controller may result in an unforeseen behaviour in the data plane. This work focuses on the SDN reactive configuration mechanism, that although originally designed for the efficient handling of changing conditions in the data plane, it can be easily misused to overload the control plane. Aiming at addressing this problem, the PDP (Programmable Data Plane)-based Controller Protection Protocol (PCPP) is presented. This protocol introduces a mechanism that efficiently filters spoofed requests at the network edge. In PCPP, end-stations require to solve a challenge before sending any connection request to the controller. The challenge answer is checked at the edge switches, which only forward valid requests to the controller. PCPP is implemented using P4, a language for programming PDP-capable devices, and its evaluation is carried out using BMv2 software switches. The results demonstrate the effectiveness of PCPP at protecting bandwidth and processing resources in the control plane against spoofed requests. A comparison against an state-of-the-art alternative not only highlights the higher efficiency of PCPP, but also its application flexibility.**

***Index Terms*—SDN, PDP, P4**

I. Introduction

Today, the influence of Software-Defined Networking (SDN) [1] has expanded from its use cases in data center or campus networks into new architectures used in other domains. Examples are next generation industrial and mobile networks, where their network architectures share SDN idea of a centralized control plane. At first glance, these new network architectures benefit of the advantages offered by SDN such as global knowledge and programmability. Nevertheless, attention must be paid to the intrinsic limitations that come with SDN. For instance, a poorly designed control plane may become a single point of failure, which under an overload situation may disrupt normal operation in the data plane.

SDN defines the network architecture presented in Fig. 1, where control and data planes are decoupled. The logically centralized and highly programmable control plane stores the application's logic, has global knowledge about its underlying resources and can be implemented in one or multiple *controllers*. The control plane is conformed by forwarding devices with no specific built-in functionality, referred as *switches*. These devices have a set of tables, that are populated with rules by the controller according to the functionality of the desired application. The *control channel* enables information exchange between control and data planes. Upon a change in the network, the switches can reactively use this channel to inform the control plane on events, that they can not handle.

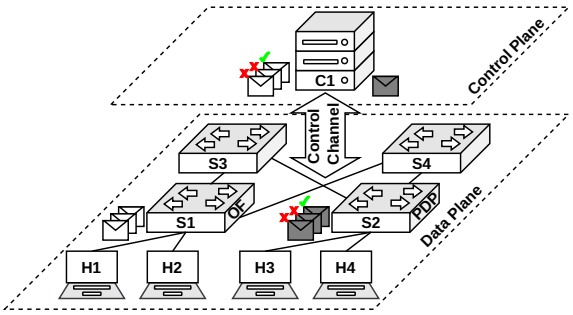


Fig. 1. SDN architecture. OF-capable switch *S1* forwards new requests, in white, to the controller for verification. While PDP-capable switch *S2* verifies new requests, in gray, on the data plane and only forwards those valid for further processing.

Reactive configuration is an efficient mechanism for handling (re)configuration requests triggered in the data plane. However, this mechanism can be easily misused by a malicious/malfunctioning end-station aiming at overloading the control plane. As depicted in Fig. 1 with *H1*, an end-station can easily generate a large number of connection requests. If these requests can not be handled by its neighboring switch *S1*, it will forward them to the controller *C1*. Thereby, the switch may not only saturate the control channel, but also exhaust processing resources in the controller. The work by Wolf et al. [2] implements the Controller Protection Protocol (CPP), which prevents controller overload in SDN networks using the OpenFlow (OF) protocol [3]. In their work, a challenge-based mechanism is used to enforce end-stations to commit resources in solving a challenge before issuing a new request, which is only processed by the controller upon answer correctness.

Unfortunately, the solution proposed by Wolf et al. presents several limitations: *i)* It is possible to overload the controller,

since it needs to verify every challenge answer. *ii)* CPP does not prevent the overload of the control channel, since every request is transmitted over this interface. *iii)* The implementation of CPP depends on TCP/IP headers, which hinders its compatibility with other networking protocols. And *iv)* challenge parameters are managed by an external entity making CPP configuration cumbersome. These limitations are derived from the OF-based implementation of CPP. However, by leveraging the capabilities of PDP these can be overcome.

The main contribution of this work is the definition, implementation and evaluation of the PDP-based Controller Protection Protocol (PCPP). The proposed protocol counts on the following features: *i)* Challenge verification is offloaded from the controller onto PDP-capable switches, hence filtering spoofed request at the network edge. *ii)* A PCPP header is designed to carry protocol information, which provides for compatibility with other networking protocols. And *iii)* challenge parameters are managed by the controller, which eases PCPP configuration. PCPP features allow to overcome the mentioned limitations in CPP. Thereby, introducing a mechanism that effectively prevents control plane overload in SDN networks by leveraging the capabilities of PDP.

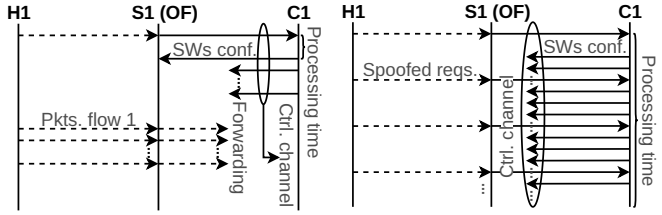
This work is structured as follows. Section II covers the problem statement and previous works. Section III describes the design and implementation of PCPP. Section IV presents the evaluation setup, experiments and results. Finally, Section V concludes this work.

II. Problem Statement & Previous Works

A. Problem Statement

Configurations in SDN usually follow a top-down approach. That is, they are triggered by applications on the control plane to be configured at the data plane. However, it is possible to react to requests in a bottom-up approach, known as a *reactive configuration*, which is depicted in the diagram of Fig. 2a. In this case, the end-station *H1* sends a packet stream for the new connection to its directly attached switch *S1*. Since this device does not know how to proceed, it forwards the first packet through the *control channel* to the controller *C1*. The controller processes the packet and decides whether configuring the connection. In case the controller decides to commit resources in the data plane, the control channel is used to deploy the configurations in the involved switches. After this, any other packets in the same flow are forwarded using the assigned resources in the data plane and no further interaction with the controller is needed.

Reactive configuration adds a mechanism for handling requests originating in the data plane. However, as depicted in Fig. 2b, this mechanism can be easily misused. For instance, a malicious/malfunctioning end-station *H1* can easily send a continuous and large amount of new requests. As a result, the following situations arise: *i) Control channel congestion*, due to the large amount packets that the switch *S1* submits to the controller *C1*. *ii) Control plane overload*, due to a large amount of incoming connection requests that the controller has to process. And *iii) data plane resources exhaustion*, since tables in switches are resources of constrained size, that can be filled if the controller decides to install rules. This last in turn contributes to introduce even more configuration traffic in the already congested control channel. The above mentioned situations can be worsened in case multiple end-stations simultaneously misuse the reactive configuration mechanism.



(a) Reactive flow configuration. (b) Control plane overload.

Fig. 2. SDN reactive configuration handling a single and a large number of requests. Dashed and solid arrows are data and control traffic, respectively.

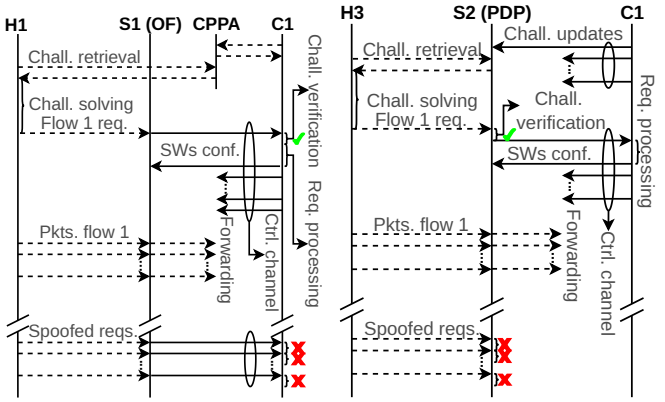
B. Previous Works

SDN architecture presents a large surface that can be targeted aiming at compromising normal network operation. The work by Eliyan et al. [4] presents a comprehensive and up-to-date survey on the topic and points out the relative simpleness how an overload situation can be triggered. In the data plane, it is possible to consume the bandwidth of a particular end-station providing a service, e.g., an HTTP server, by repeatedly sending a large amount of spoofed requests. Nevertheless, compared to traditional networks, SDN networks are better suited to cope with this situation as demonstrated in ORACLE [5], where SDN, PDP and Machine Learning (ML) are combined for the detection of Denial-of-Service (DoS) attacks in the data plane.

More dangerous than attacks that target an specific end-station are those that target the control plane by misusing the reactive configuration mechanism. Works covering control plane overload can be categorized in *detection*, *mitigation*, and *prevention*. Detection includes those works that simply identify whether an overload situation is taking place and can be further divided into *metrics* [6] or *ML-based* [7]. Metric-based detectors employ statistical measurements, e.g., packet rate, or Instruction Detection Systems (IDS), while ML-based methods train complex models using multiple data plane metrics. Mitigation works go a step beyond after the identification and try to correct the situation. These rely either in blocking ports for end-stations with high packet rates [8] or reflecting packets to a storage [9]. The prevention category contain those works, that protect the controller by adding a filtering layer between the data and control planes [10]. A comprehensive study on the effects of control plane overload for multiple commercial controllers is presented in [11].

The work presented by Wolf et al. [2] adds another dimension in the prevention category by offloading almost all processing needed to prevent a control plane overload to the end-stations. For that, the controller challenges end-stations

and only processes their requests upon answer correctness. The authors leverage the capabilities of Proof-of-Work (PoW) [12] for the implementation of CPP. As depicted in Fig. 3a, when the end-station *H1* wishes to establish a new connection, it first retrieves the current challenge from the CPP Authority (CPPA). Then, it solves the challenge and sends its answer along with the first packet in the flow to its neighboring switch *S1*. This device forwards the packet over the control channel to the control plane. The controller *C1* retrieves the challenge from the CPPA and verifies the answer. Upon correctness, the resources in the data plane are configured, otherwise the request is dropped.



(a) OF-based Controller Protection Protocol (CPP).

(b) PDP-based Controller Protection Protocol (PCPP).

Fig. 3. Operation of state-of-the-art CPP and proposed PCPP. Dashed and solid arrows represent data and control traffic, respectively.

III. Design & Implementation

A. PDP-based Controller Protection Protocol (PCPP)

SDN reactive configuration is an efficient mechanism for handling (re)configurations triggered by changing conditions in the data plane. However, it can be easily misused by malicious/malfunctioning end-stations aiming at overloading the control plane. This work proposes the PDP-based Controller Protection Protocol (PCPP), a mechanism that allows to leverage the benefits of SDN reactive configuration without exposing the network to a potential control plane overload.

Fig. 3b depicts the operation of PCPP. When the end-station *H3* wishes to establish a new connection, it first retrieves the challenge information from its neighboring PDP-capable switch *S2*. Then, it solves the challenge and sends the flow's first packet along with the answer. The switch *S2* verifies the answer and only upon its validity, forwards the packet over the control channel to the controller *C1*. In the controller, the connection request is processed and in case it can be attended, the respective resources in the data plane are provisioned. Once the resources are configured, all subsequent packets in the flow are forwarded in the data plane without further interaction with the control plane. In the event that an end-station sends a large amount of spoofed requests, as depicted in the lower part of Fig. 3b, these can be easily detected and dropped at the network edge. Thus, preventing the overload of the controller and the control channel.

B. End-stations

According to PCPP operation depicted in Fig. 3b, end-stations are in charge of *flow generation*, *challenge retrieval*, and *challenge answering*. Flow generation corresponds to common traffic generation. Thus, focus will be only given to the last two tasks, as these are the new steps required when setting up a connection.

1) Challenge retrieval: In PCPP, end-stations retrieve challenges from their neighboring switch as depicted in the upper part of Fig. 3b. For that, an end-station sends a packet with the headers as shown in Tbl. I. Fields inside the PCPP header are described in Tbl. II. During challenge retrieval, values inside the PCPP header are irrelevant. Once the packet reaches the neighboring switch, this device populates the current challenge information in the PCPP header, clears out the answer field and bounces the packet back towards the requesting end-station.

TABLE I
PCPP PACKET HEADER ORDERING WITH OPTIONAL HEADERS IN GRAY.

L2 Ethernet hdr.	PCPP hdr.	L3 IP hdr.	L4 TCP/UDP hdr.	Payload

TABLE II
PCPP HEADER STRUCTURE WITH TOTAL LENGTH OF 128 BITS.

EtherType (16 bits)	Solution (16 bits)
Challenge (32 bits)	
Answer (64 bits)	

2) Challenge answering: Challenges in PCPP are implemented using the principles of PoW. This mechanism can be used to verify whether an user committed resources before issuing a request to a server. In this case, an user invests computing resources in answering a challenge. The challenge answer is sent along with the request to the server and will only be processed upon answer correctness. In basic PoW, the server selects a function *f*, and the time changing parameters: challenge c_t and solution s_t, as described by Eq. (1). An end-station willing to do a request to this server, has to find the answer a_t, such that after plugging c_t and a_t into function *f*, the results yields s_t. The function *f* is fundamental in PoW and any candidate should ensure the following features: *i)* Finding a_t, i.e., answering the challenge, must be a time consuming operation. *ii)* Computing s_t, i.e., verifying the answer, must be a quick operation. And *iii)* the time to find a_t depend on the election of parameters c_t and s_t.

$$s_t = f(c_t, a_t) \tag{1}$$

Hash functions are candidates for function f due to the following properties: *i) Fixed-length output*, an input of any arbitrary length is mapped into fixed-length output, known as

a *digest*. *ii) Pre-image resistance*, it is computationally expensive to find the input producing a given digest. *iii) Collision resistance*, it is hard to find two different inputs that produce the same digest. And *iv)* Hash are *unbiased* functions, i.e., a Hash should yield an unbiased digest even if the given inputs are biased. When Hash functions are used in PoW, the solution s_t is referred as challenge *complexity* and represents the count of leftmost continuous zeros in the digest. The task for an end-station willing to establish a new connection is to find the answer a_t, that concatenated with the challenge c_t produces a digest with a least s_t heading zeros.

PoW implementation in CPP by Wolf et al., uses the SHA Hash function. Unfortunately, PDP devices do not support algorithms to compute such Hash function. However, Cyclic Redundancy Check (CRC) functions represent an alternative for Hash functions in PCPP implementation. CRC functions are generally implemented in forwarding devices for error detection in data transmission. These functions share most of the properties of Hash functions. For instance, a CRC-n maps an arbitrary length input to a fixed n-bit length output, known as checksum. Despite the similarities between Hash and CRC functions, they should not be used interchangeably, due to the fact that CRC functions are not designed to be unbiased, which limits its application in data integrity applications. Nevertheless, CRC can be used for PoW realization in PCPP, since data integrity does not play a role in the protocol and it fulfills with the requirements for challenge function *f*.

$$s_t = NLZ\_CRC(c_t, p, a_t) \tag{2}$$

Eq. (2) describes PoW implementation in PCPP. Function f is realized using the function NLZ_CRC of Alg. 1, which returns the count of continuous leftmost zeros in the checksum. The operator $||$ represents bit concatenation and the variable p represents layer dependant connection parameters, as described in Tbl. III. The parameter p is required to uniquely identify a connection request that an end-station wishes to establish. Without p, an end-station could solve the current challenge c_t and use its answer a_t to trigger multiple new connection requests. However, by concatenating the challenge c_t with some connection parameters p, the end-station has to solve a unique composed challenge $c_t||p$ per connection request, although the challenge c_t is the same.

TABLE III
CONNECTION PARAMETERS p ACCORDING TO THE LAYER.

Layer	p	Bit cnt.
2	Src. MAC || Dst. MAC	96
3	Src. IP || Dst. IP	64
4	Src. IP || Dst. IP || Proto. || Src. Port || Dst. Port	104

Alg. 1 also describes the function DO_WORK, which is used by end-stations to solve the composed challenge. Since a CRC is hardly reversible, the algorithm relies in a *brute-force* search to find the answer a_t, producing a checksum s' starting with a least s_t zeros. The bit-width for the parameters solution s_t, challenge c_t and answer a_t correspond to the size of their respective fields in the PCPP header definition of Tbl. II. Connection parameters p are implicit in the connection information contained in the layer 2-4 headers.

Algorithm 1 Proof-of-Work (PoW) implementation in PCPP.

1: **function** NLZ_CRC(c_t, p, a_t)
2: $checksum \leftarrow CRC(c_t||p||a_t)$ ▷ $||$ bit concatenation
3: **return** $count\_num\_leftmost\_zeros(checksum)$
4: **function** DO_WORK(s_t, c_t, p)
5: $s' \leftarrow 0, a_t \leftarrow 0$
6: **while** $s' < s_t$ **do** ▷ At least s_t zeros
7: $a_t \leftarrow random\_number()$
8: $s' \leftarrow NLZ\_CRC(c_t, p, a_t)$
9: **return** a_t

C. Challenge Hardness Analysis

Due to a CRC is a hardly reversible function, the implementation of DO_WORK in Alg. 1 reduces to a brute-force search for the answer a_t. Assuming that any random answer a_t can generate with equal probability any checksum, then the probability of having at least s_t consecutive zeros can be seen as a sequence of Bernoulli trials with a fair coin as described in Eq. (3), where the random variable *Y* represents the count of leftmost consecutive zeros in the checksum. The probability of having at least s_t consecutive zeros is equal to the probability of having exactly s_t consecutive zeros, since these probabilities are not exclusive. For instance, the probability of having a zero is a half, and it already contains the probability for other sequences containing at least a consecutive zero.

$$P(Y \geq s_t) = P(Y = s_t) = 2^{-s_t} \tag{3}$$

The number of tries X an end-station needs to perform before finding the desired checksum can be related to number tosses required before getting s_t heads, or zeros in this case. This is formulated in Eq. (4), where $E[X] = \bar{x}$ is the expected number of tries. The first term in the expression represents that after the first try a one appeared, thus an additional try is needed. The second term represents that a zero followed by a one appeared, thus an additional try is needed. The last term represents that the sequence has s_t consecutive zeros. For example, if only a zero is required $s_t = 1$, then the expression in Eq. (4) can be reduced to its first and last term. In this case, on average two tries are required, i.e, $E[X] = \bar{x} = 2$, before finding the desired checksum.

$$\bar{x} = 2^{-1}(\bar{x}+1)+2^{-2}(\bar{x}+2)+\ldots+2^{-s_t}(\bar{x}+s_t)+2^{-s_t}s_t \tag{4}$$

After solving Eq. (4) with help of the geometric series, $\bar{x}$ can be expressed as the first equality in Eq. (5). If $s_t >> 1$, $\bar{x}$ can be approximated as an exponential function of $s_t + 1$. Moreover, if the decimal logarithm is applied to this approximation, the result in Eq. (6) is reached. In this last approximation,

it is evident that the exponent of the expected number of tries grows proportionally with the solution parameter s_t. For instance, when a checksum with at least ten zeros is required $s_t = 10$, on average the end-station has to perform about 10^3 tries. From the results in Eqs. (3) and (6), it is clear how the parameter s_t can be used in PCPP to manage the challenge complexity. As by increasing the number of desired zeros s_t, the probability of finding an answer exponentially drops and the number of expected tries exponentially grows.

$$\bar{x} = 2(2^{s_t} - 1) \approx 2^{s_t+1} \tag{5}$$

$$log_{10}(\bar{x}) \approx log_{10}(2)(s_t + 1) \tag{6}$$

D. Switches

The pipeline of PDP-capable switches in PCPP is described in Fig. 4, where the following three tables, depicted as gray blocks, are implemented:

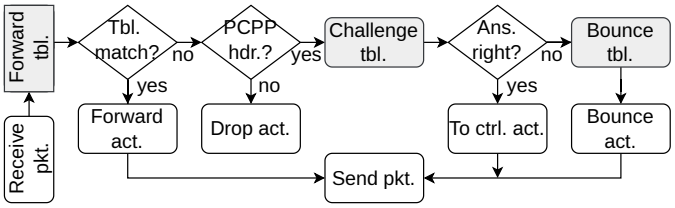


Fig. 4. PCPP switch pipeline with actions (act.) and tables (tbl.) in grey.

1) Forwarding table: When a switch in PCPP receives a packet, it is first matched against the *Forwarding table*. This table performs layer 2-4 forwarding according to the rules installed by the controller. Upon a match the *Forward action* selects the proper output port, strips the PCPP header, if present, and proceeds to send the packet.

2) Challenge table: In case there was not a match in the *Forwarding table*, the switch checks the existence of the PCPP header. If absent, the packet is dropped using the *Drop action*. If present, the packet is sent to the *Challenge table*. Here, the switch verifies if the answer provided in the PCPP header produces the desired number of consecutive zeros s_t, as described in the function NLZ_CRC of Alg. 1. For that, the current challenge c_t is combined with the connection parameters p extracted from the layer 2-4 headers and the provided answer a_t. Then, the checksum is computed. If the answer a_t is correct, the *To controller action* is invoked, which strips the PCPP header and sends the packet to the controller.

3) Bounce table: If the provided answer a_t was incorrect, the packet is sent to the *Bounce table* and the *Bounce action* is invoked. This action clears out the the answer field in the PCPP header and populates the fields for the challenge c_t and solution s_t with their current values. Then, source and destination addresses are swapped and the packet is sent out towards its originating end-station.

E. Controller

The main functions of the controller in PCPP include *challenge distribution* and *routing*. These are next explained.

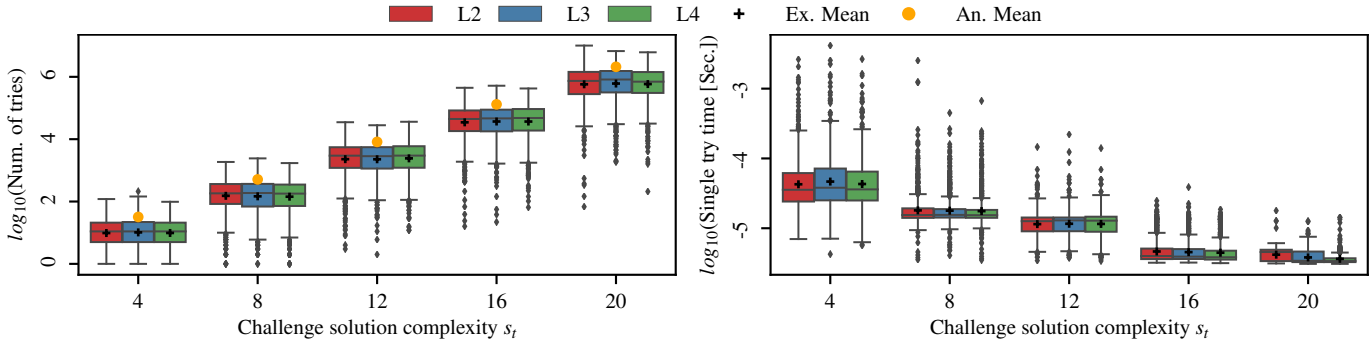
1) Challenge distribution: This function consists on the periodical update of challenges c_t and solutions s_t in the switches. For each switch, the controller periodically defines a random challenge c_t and its associated solution complexity s_t. Then, as depicted in Fig. 3b, over the control channel the controller deploys the configurations in the switches by replacing previous entries in the *Challenge table* with the new values. Although end-stations attached to the same switch share the same challenge information, it is still possible to guarantee that they solve different challenges. Because each end-station solves the composed challenge $c_t||p$, where p contains information proper of the being connection requested.

In PCPP, there are two situations when it is possible for an end-station to stockpile correct challenge answers, that could be later used to trigger an overload situation. *i)* When the update interval *T* is too long, which generates large time windows with the same challenge information in the switches. PCPP uses a short update interval under a minute to avoid this situation. Moreover, the update interval can be changed at runtime to shorten this window even further according to the network dynamics. And *ii)* when the challenge complexity is too low. Hence, end-stations can find answers quickly. PCPP offers the possibility to increase challenge complexity on the fly with a switch granularity to deal with this situation. In PCPP, the solution parameter s_t can have a complexity range between 1 to N zeros, where N is given by the length of the checksum. As it will be shown in the upcoming Section IV, this allows to have answering times ranging from less than a second, to minutes and even hours.

2) Routing: This function comprehends path computation and its deployment in the data plane. Once a packet with a valid answer a_t is verified by the edge switch, this device strips the PCPP header and sends the connection request to the controller over the control channel. Then, the controller checks whether for the requested source and destination a path exists. If so, the connection is provisioned by adding the necessary rules in the *Forwarding table* of the switches involved in the path. Otherwise, the controller ignores the connection request.

F. Protocol Applicability

The proposed PCPP is a mechanism that prevents control plane overload in SDN networks and hence should not be confused with a detection mechanism. PCPP is conceived to be deployed in networks handling untrusted end-users, where the controller re-actively processes user requests as in Internet Service Provider (ISP) or campus networks. Current PDP-capable switches and end-stations have all the functionalities needed to realize PCPP. Besides, the modifications required at end-stations are changes at the application level, which can be done in software. PCPP could be used in a *hybrid* network, i.e., a network with some PDP-capable switches. In that case, only the PDP-capable switches verify challenges and the non-PDP switches could offload this operation on them. PCPP exhibits low overhead and high scalability. Its overhead is attributed to the verification at the switches and the protocol configuration. This last is managed by the controller and its parameters can



(a) Analytical (An.) and experimental (Ex.) number of tries before finding the challenge answer a_t, as solution complexity s_t increases and layer changes.

(b) Experimental try time in seconds for computing a single candidate answer, as solution complexity s_t increases and layer changes.

Fig. 5. Fig. 5a shows how as the solution complexity s_t increases, the number of tries needed to find an answer a_t exponentially grows, while within the same complexity, the connection layer does not introduce any variation. Fig. 5b depicts the average time spend computing a single candidate answer. As the solution complexity s_t increases, the average answering time drops, while within the same complexity, the connection layer does not introduce any variation.

be tuned according to the network dynamics. PCPP scalability does not depend on the network size. In fact, the larger the network, the greater the performance improvement, as the processing resources and bandwidth saved when preventing an overload, exceed by far PCPP overhead.

IV. Evaluation & Results

This section first introduces the setup used for evaluating PCPP. Then, the experiments and their results are presented.

A. Evaluation Testbed

In PCPP, PDP is realized using Programming Protocol-independent Packet Processors (P4) [13], a domain-specific language for programming data plane functionality in forwarding devices such as SmartNICs and software or hardware switches. In the evaluation setup of PCPP, the Behavioral Model version 2 (BMv2) software switch has been selected, since with this switch is possible to easily define a test environment for the validation of P4 applications. Using the network emulator Mininet [14], the data plane shown in Fig. 1 was implemented. The combination of Mininet, BMv2 and OF software switches allow to obtain results corresponding to the ones expected in a real network, which would not be possible with a simulation. Depending on the evaluation scenario, switches *S1-S4* can be either BMv2 software or OF-capable switches. Virtual end-stations *H1-H4* are executed inside their own namespace. The control plane consists of the controller *C1*, which is written as a Python application. Both control and data planes run inside their own Virtual Machine (VM). Each VM counts on 4 cores and 8192 MB of RAM. Communication between the VMs hosting control and data planes is possible over the control channel, which is realized using the P4Runtime protocol [15].

The BMv2 switch counts on a limited selection on CRC functions, which includes CRC-32 [16]. Due to its 32-bit checksum, the CRC-32 was selected as the challenge function f for the realization of function NLZ_CRC in Alg. 1. This gives the solution complexity s_t a range between 1 to 32 zeros.

TABLE IV
Mean values for: number of tries, single try time, controller and switch verification times, depicted in Figs. 5a, 5b, 6a and 6b, respectively. Mean values for each variable at each complexity were averaged across the connection layers.

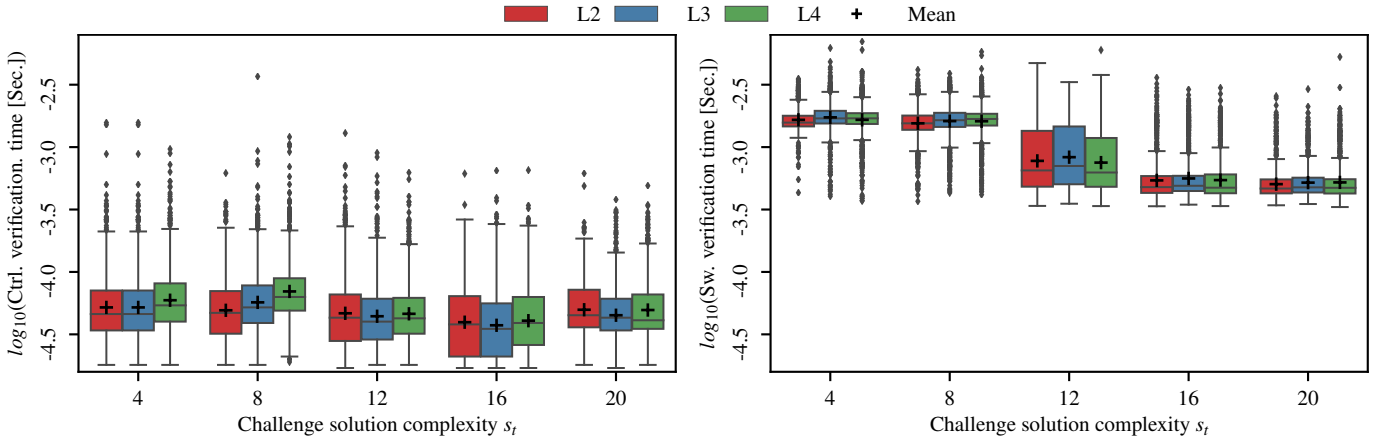
Complexity	4	8	12	16	20
Mean An. num. tries	$10^{1.5}$	$10^{2.7}$	$10^{3.9}$	$10^{5.1}$	$10^{6.3}$
Mean ex. num. tries	10^{1}	$10^{2.2}$	$10^{3.4}$	$10^{4.6}$	$10^{5.8}$
Mean single try time [s]	$10^{-4.4}$	$10^{-4.7}$	$10^{-4.9}$	$10^{-5.3}$	$10^{-5.4}$
Mean ctrl. verif. time [s]	$10^{-4.3}$	$10^{-4.2}$	$10^{-4.3}$	$10^{-4.4}$	$10^{-4.3}$
Mean sw. verif. time [s]	$10^{-2.8}$	$10^{-2.8}$	$10^{-3.1}$	$10^{-3.3}$	$10^{-3.3}$

B. Challenge Answering

In this first experiment, the performance of PCPP at the end-station side is evaluated. For that, the end-station *H1* in Fig. 1, selects depending on the layer random destination information to create the connection parameter p, solves the current composed challenge $c_t||p$, and sends out the new connection request along the answer a_t to its neighboring P4-capable switch *S1*. This experiment is performed for layer 2 to 4 and solution complexities $s_t = (4, 8, 12, 16, 20)$. For each layer and complexity a batch of 1000 samples is evaluated.

Figs. 5a and 5b present the results for this experiment. In particular, Fig. 5a displays the experimental number of tries end-station *H1* has to perform before finding the challenge answer a_t as a function of the challenge complexity and the connection layer. As the solution complexity s_t increases, the number of tries exponentially grows. According Tbl. IV, the experimental mean number of tries increases from 10^1 at $s_t = 4$, to approximately 10^6 tries at $s_t = 20$. Recalling the approximation for the analytical mean number of tries given in Eq. (6), the exponential increase of the number of tries with respect to the complexity was expected. The result of this approximation is depicted in Fig. 5a using yellow circles.

The difference between the analytical and experimental



(a) Experimental time the controller spends verifying the challenge answer a_t, as the solution complexity s_t increases and the layer changes.

(b) Experimental time a software BMv2 switch spends verifying the challenge answer a_t, as the solution complexity s_t increases and the layer changes.

Fig. 6. Figs. 6a and 6b describe the challenge answer a_t verification time for the controller and the BMv2 software switch, respectively. In the controller, the verification time remains approximately constant across all solution complexities s_t, whereas in the BMv2 switch, it takes longer for lower solution complexities s_t and drops at higher complexities. In general, the controller performs verification faster than the BMv2 switch and the change in connection layer introduce only subtle variations.

means may be attributed to two factors. *i)* The mathematical inaccuracy introduced by the approximation. And *ii)* the assumption done to derive the analytical mean number of tries. This states that any random answer a_t can generate with equal probability any checksum, which may not be completely true in the case of CRC functions. A difference of Hash functions, a CRC is not designed to be an unbiased function. This implies that a CRC may produce similar checksums for biased inputs, which in turn may favor certain answers. Hence, reducing the number of iterations before finding a right challenge answer.

Fig. 5b presents the time end-station *H1* needs to perform a single try for finding a candidate answer. As described also in Tbl. IV, at lower solution complexities s_t, the mean single try time takes longer and it decreases as the solution complexity increases. For instance, at $s_t = 4$ a single try takes on average $40\mu s$, whereas at $s_t = 20$ it takes $4\mu s$. At lower solution complexities, the end-station *H1* requires less time to find the challenge answer a_t before creating and sending packets. These operations require processing resources and can interfere with the answer search for the subsequent challenges. At higher complexities, this interference does not play a role, since packet transmission is less frequent. This due to the larger amount of tries needed before finding the challenge answer, despite a single try takes less time. For instance, for $s_t = 4$ a packet is on average ready for transmission each $0.4ms$, whereas with $s_t = 20$ it is ready only after $40s$.

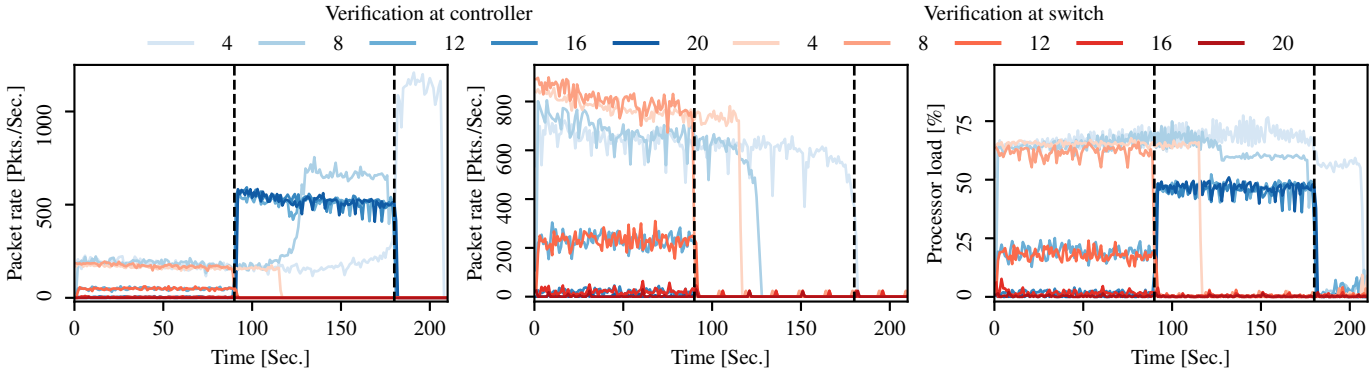
From the previous results, it is clear how the solution complexity s_t of PCPP can be used to control the number of tries an end-station performs before finding a challenge answer a_t, which indirectly allows to manage the time and end-station spends solving a challenge. Eq. 6 can be used to extrapolate the upper limit on average number of tries for scenarios with higher solution complexities. For instance, when $s_t = 28$, $11^{8.7}$ tries are expected and it would take, to an end-station with the same resources of *H1*, 33 minutes on average.

C. Challenge Verification

The goal with this experiment is to evaluate and compare challenge verification performance in the controller and the BMv2 software switch. This provides a baseline comparison between the state-of-the-art CPP by Wolf et al. [2], where verification is carried out in the control plane, against PCPP, where verification is performed at switches in the data plane. For this, an experimental setup based on the topology of Fig. 1 is used. In this case, end-stations *H1* and *H3*, send batches of 1000 connection requests, gradually increasing the solution complexity $s_t = (4, 8, 12, 16, 20)$ and changing the connection layer from 2 to 4. End-station *H1* is attached to the OF-capable switch *S1*, which forwards connection requests over the control channel to the controller *C1* for verification. In comparison, end-station *H3* is attached to the P4-capable switch *S2*, which is able to verify challenge answers itself.

Fig. 6a depicts the results when challenge verification is performed in the controller *C1*. From the figure and Tbl. IV, it is clear that challenge verification takes approximately $50us$ regardless of the increase in complexity s_t. Moreover, for a particular challenge complexity connection layer 2 and 4 tend to require slightly more time. The approximately constant time for challenge verification can be attributed to the low load in the controller. For example, in the scenario with the highest possible packet inter-arrival rate, i.e., when $s_t = 4$, a packet arrives at the controller each $0.4ms$, which gives *C1* a large time window to verify the challenge before the next packet arrives. The subtle variation caused by the connection layer is derived from the length of connection parameter p. As shown in Tbl. III, layers 2 and 4 involve longer headers fields, which results in the challenge verification operation being carried out over slightly more bits for those layers.

Fig. 6b depicts the results when challenge verification is performed in the BMv2 software switch *S2*. As depicted, the verification time drops as the solution complexity s_t



(a) Packet-in rate in control channel. (b) Packet-out rate in control channel. (c) Controller processor utilization.

Fig. 7. Performance comparison for normal and overload situations when challenge answers are verified at the controller or BMv2 software switches, depicted with blue and orange lines, respectively. Dashed vertical lines in the figures at $t = 90s$ and $t = 180s$ denote the end of normal and overload situations.

increases, while for the same complexity the connection layer does not add a significant variation. According to Tbl. IV, at lower solution complexities i.e., $s_t = 4$ or 8, the verification time takes on average $1.6ms$, whereas at higher solution complexities i.e., $s_t = 16$ or 20, the verification takes $0.5ms$. The higher verification time at lower complexities is due to the frequent arrival of packets in *S2*. For instance, when $s_t = 4$ the packet inter-arrival time is $0.4ms$. This results in packets being queued, since *S2* processes packets at a lower rate. Thus, incoming packets interfere with the resources available for challenge verification. On the contrary, at higher complexities, packets arrivals are less frequent. For example when $s_t = 20$ the inter-arrival time is $40s$, which gives *S2* plenty of time to verify challenges without any interference.

In the presented evaluation, challenge verification at the controller was about 10 times faster than at the BMv2 software switch. However, this neglects the fact, that controller *C1* had at its disposition all resources inside its VM, whereas switch *S2* shared the resources of its VM with all other components in the data plane. Besides, the BMv2 software switch is a tool for development of P4 applications, which is not meant to be used as production-grade switch [17]. Hence, the results obtained with the BMv2 software switch represent an upper limit on the achievable challenge verification time with a P4-capable device such as hardware switches or SmartNICs.

D. Control Channel Load and Controller Utilization

In this experiment, the performance of PCPP and the state-of-the-art CPP presented by Wolf et al. [2] are evaluated under normal and overload situations. For that, challenge verification at the controller and the BMv2 software switches are compared in terms of *packet-in*, *packet-out rates* and *controller processor utilization*. The respective results are depicted in Figs. 7a, 7b and 7c. Packet-in are packets that the controller receives from the control channel, whereas packet-out are packets that the controller sends into the control channel. Dashed vertical lines delimit the end of normal and overload operation at $t = 90s$ and $t = 180s$. For verification at the controller shown with blue lines, the data plane depicted in Fig. 1 consists only of OF-capable switches. During normal operation, the end-station *H1* requests valid random layer 4 connections for $90s$ to end-stations *H2,H3* and *H4*. After this, the overload situation starts, where the end-station *H1* sends spoofed connection requests as fast as possible. This process is repeated for solutions complexities $s_t = (4, 8, 12, 16, 20)$. For verification at the switches shown with orange lines, the data plane consists only of BMv2 software switches. In this case, the same procedure is repeated, but *H3* becomes the end-station generating the connection requests with *H1,H2* and *H4* as destinations.

From Figs. 7a, 7b and 7c it is evident, that lower challenge complexities, i.e., $s_t = 4$ or 8, add unpredictability in the results. For instance, in Fig. 7a for verification at the controller with challenge complexity $s_t = 4$, a burst of packet-in is received, even after packet generation has ended at $t = 180s$. This is caused by the short packet inter-arrival time of $0.4ms$ at this complexity, which results in a large amount of valid connection requests during normal operation. These requests are queued in the data plane and when the overload situation starts invalid requests are added at an even higher rate in the queue. Figs. 7b and 7c show that during normal operation, the controller commit resources in processing the valid requests and deploying their configurations using packet-out messages. During the overload situation, enqueued valid requests still reach the controller and it continues processing and deploying their configurations. Before $t = 180s$ the controller finishes processing the enqueued valid requests. This causes a drop in the processor load and no more packet-out are sent. In turn, this frees resources in the emulated control channel and trigger the burst of invalid packet-in requests, for which the controller commits resources in their verification until $t = 210s$. For verification in the switches with lower challenge complexity, i.e., $s_t = 4$, a similar situation happens. However, since spoofed requests are filtered a the network edge, the controller is able to conclude configuration deployment at $t = 120s$, i.e., $60s$ earlier than when verification is done at the controller.

At higher challenge complexities, i.e., $s_t = 12, 16$ and 20, connection requests inter-arrival times are longer, which

filters out the effects of queuing in the results. As depicted in Figs. 7a, 7b and 7c during normal operation, regardless of where verification takes place and according to the complexity, a similar amount of packet-in requests reaches the controller, which triggers a similar demand in processing resources and packet-out configuration messages. However, during the overload situation, the benefits of PCPP over state-of-the-art CPP are evident. In the latter, the controller receives spoofed requests at the rate that the end-station can produce them. In this case, it was about 520 packets per second. However, this rate can become the sum of the individual generation rates, if not one, but multiple end-stations generate simultaneously spoofed requests. For challenge verification at the switches, the controller receives zero packet-in requests during the overload scenario, since spoofed requests are filtered at the edge by the switches. As there are not incoming packet-in messages, the controller in PCPP does not spend processing resources in verifying challenge answers, as depicted in Fig. 7c. Whereas for verification at the controller, around 45% of its processing resources are spent in checking spoofed requests. As shown in Fig. 7b, under overload none of the alternatives deploy configurations for spoofed requests. The periodic bursts in the packet-out rate and the processor load, for verification at the switches, correspond to PCPP operations for challenge update in the switches. The protocol overhead depends on the selected update interval and the amount of switches in the data plane. In the evaluation, this interval was fixed to $T = 15s$ and on average only 2 PCPP update packets were sent each second.

V. Conclusions

This work proposes the PDP (Programmable Data Plane)-based Controller Protection Protocol (PCPP), which leverages the benefits PDP to prevent control plane overload caused by misusing Software-Defined Networking (SDN) reactive configuration mechanism. In PCPP, end-stations solve a challenge before issuing a connection request. The challenge is verified at PDP-capable switches, which only forwards the request to the controller upon validity. Hence, filtering out spoofed requests at the network edge. PCPP challenge complexity can be used to manage the number of tries required by end-stations before answering a challenge, which gives an indirect handle on the time needed at end-stations to perform requests, ranging from less than a second to minutes and even hours. The implementation of PCPP demonstrates how the protocol is realized using P4 devices and the Cyclic Redundancy Check (CRC) function. The evaluation presents a comparison between PCPP and a state-of-the-art alternative in terms of challenge verification time, control channel load and controller utilization. Although verification required 10 times more time at BMv2 software switches than at the controller, this represents rather an upper bound on the achievable verification time with a P4 device. Moreover, the benefits of PCPP are notorious in savings of bandwidth and processing resources, when an overload situation is prevented, since neither bandwidth in the control channel nor processing in the controller are invested. This represents savings of 100% in these resources with PCPP, when compared with the state-of-the-art alternative performing verification at the control plane, where the controller receives as many connection requests as the end-stations can generate and thus invests a proportional amount of bandwidth and processing resources. As future work, remains the evaluation PCPP using P4 hardware switches and SmartNICs, where a faster challenge verification is expected.

Acknowledgment

This work is partially funded by Federal Ministry of Education and Research in Germany (BMBF) as part of the project AI-NET-ANTILLAS (grant ID 16KIS1318).

References

[1] ONF. Software-Defined Networking (SDN) Definition. Accessed: May 13, 2022. [Online]. Available: https://bit.ly/3KQO7Zp

[2] T. Wolf and J. Li, "Denial-of-service prevention for software-defined network controllers," in *2016 25th Int. Conf. on Computer Communication and Networks (ICCCN)*, 2016, pp. 1–10.

[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, p. 69–74, mar 2008.

[4] L. F. Eliyan and R. Di Pietro, "DoS and DDoS attacks in software defined networks: A survey of existing solutions and research challenges," *Future Generation Computer Systems*, vol. 122, pp. 149–171, 2021.

[5] S. G. Macías, L. P. Gaspary, and J. F. Botero, "ORACLE: An Architecture for Collaboration of Data and Control Planes to Detect DDoS Attacks," in *2021 IFIP/IEEE Int. Symposium on Integrated Network Management (IM)*, 2021, pp. 962–967.

[6] A. A. Y. R. Fares, F. L. de Caldas Filho, W. F. Giozza, E. D. Canedo, F. L. Lopes de Mendonça, and G. D. Amvame Nze, "DoS Attack Prevention on IPS SDN Networks," in *2019 Workshop on Communication Networks and Power Systems (WCNPS)*, 2019, pp. 1–7.

[7] T. Abhiroop, S. Babu, and B. S. Manoj, "A Machine Learning Approach for Detecting DoS Attacks in SDN Switches," in *2018 Twenty Fourth National Conf. on Communications (NCC)*, 2018, pp. 1–6.

[8] N. T. Tran, T. L. Le, and M. A. T. Tran, "ODL-ANTIFLOOD: A comprehensive solution for securing opendaylight controller," in *Int. Conf. on Advanced Computing and Applications (ACOMP)*, 2018, pp. 14–21.

[9] J. S. Maddu, S. Tripathy, and S. K. Nayak, "Sdnguard: An extension in software defined network to defend dos attack," in *2019 IEEE Region 10 Symposium (TENSYMP)*, 2019, pp. 44–49.

[10] S. Y. Khamaiseh, A. Al-Alaj, and A. Warner, "FloodDetector: Detecting Unknown DoS Flooding Attacks in SDN," in *2020 Int. Conf. on Internet of Things and Intelligent Applications (ITIA)*, 2020, pp. 1–5.

[11] A. A. Alashhab, M. Soperi Mohd Zahid, A. A. Barka, and A. M. Albaboh, "Experimenting and evaluating the impact of DoS attacks on different SDN controllers," in *2021 IEEE 1st Int. Maghreb Meeting of the Conf. on Sciences and Techniques of Automatic Control and Computer Engineering MI-STA*, 2021, pp. 722–727.

[12] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *Advances in Cryptology — CRYPTO' 92*, E. F. Brickell, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 139–147.

[13] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[14] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: Association for Computing Machinery, 2010.

[15] ONF. P4Runtime Specification. Accessed: May 13, 2022. [Online]. Available: https://bit.ly/3KRGPV4

[16] ONF. P4-16 declaration of the P4 v1.0 switch model. Accessed: May 13, 2022. [Online]. Available: https://bit.ly/3x4uIyh

[17] ONF. Performance of BMV2. Accessed: May 13, 2022. [Online]. Available: https://bit.ly/3RCKSGU