PathSafe: Secure Path Verification in Software-Defined Networks

Doriana Monaco*, Nikola Antonijević†, Sayon Duttagupta†, Dave Singelée†, Alessio Sacco*, Eduard Marin‡, Bart Preneel†
*DAUIN, Politecnico di Torino, Italy
†COSIC, KU Leuven, Belgium
‡Telefónica Research, Spain

***Abstract*—Network topology verification in Software-Defined Networks (SDN) poses a significant challenge, as vulnerabilities can allow attackers to deceive the controller and manipulate the data plane into incorrect topologies, thereby endangering the entire network's security. Current solutions fail to guarantee both security and efficiency in the verification process, often resulting in damaging user traffic. With the aim of solving joint objectives, in this paper, we introduce *PathSafe*, a novel tool constructed on top of the existing controller frameworks designed for secure path verification in SDN environments. It enables the verification of all available paths between two points in the network and ensures a secure process. Our approach requires a data plane component for real-time packet monitoring at line speed and a control plane verification step. Our research demonstrates that *PathSafe* effectively mitigates security risks in compromised switches and host scenarios. Alongside a theoretical exploration of this challenge, we present a proof of concept implemented in P4, a common language for programmable data planes. Results obtained in Mininet underscore the practical applicability of *PathSafe* that, compared to alternatives, reduces overhead in the verification process while maintaining a limited execution time.**

***Index Terms*—Software-Defined Networks, P4, Path Verification, Security Services**

I. Introduction

Software-Defined Networks (SDN) have gained significant momentum in enabling more dynamic, agile and programmable networks in data center, cloud, telecommunications and enterprise environments [1]. SDN advocates for the decoupling of the network's intelligence (i.e., the control plane) from its data forwarding functions (i.e., the data plane). The SDN controller manages core services essential for critical network functions, such as routing and topology discovery. Through SDN applications that operate on top of the controller and a standard southbound interface like OpenFlow [2], the SDN controller can easily reprogram and collect statistics from networking devices. These applications enable network operators to implement high-level networking tasks, as well as security and privacy applications [3], [4], without altering the SDN controller's underlying logic. However, despite its advantages, SDN also expands the network's attack surface and introduces new security challenges. Additionally, due to fundamental differences between traditional and SDN architectures, existing security solutions for conventional networks cannot be directly applied to SDN environments [5].

In this paper, we focus on topology attacks, which seek to compromise the controller's view of the network topology and are especially dangerous in SDN-based networks. Recent research has revealed that SDN topology services lack sufficient security mechanisms [6]–[8]. Attackers have exploited the weaknesses in the Link Layer Discovery Protocol (LLDP) [9] to tamper with the controller's view of the network topology to perform eavesdropping, Man-In-The-Middle (MiTM) attacks or bypassing middleboxes [10], [11]. The consequences of these attacks can be severe, as SDN core services and applications rely on accurate topology information to function correctly.

To address these issues, various solutions have been proposed, which can be divided into two groups: (i) those that attempt to fix the identified security problems by modifying the LLDP protocol implemented within the SDN controller (e.g., [6], [8], [12]), and (ii) those that propose alternatives to the LLDP protocol to be included within SDN controllers (e.g., [13], [14]). However, both types of solutions suffer from drawbacks. Firstly, directly implementing changes or new solutions within the SDN controller is undesirable, as it may introduce new security vulnerabilities into the 'brain' of the network. Moreover, these solutions are unlikely to be adopted in practice, as evidenced by the fact that none of the well-known open-source SDN controllers have integrated any of the proposed countermeasures to defend against topology attacks. Secondly, none of these solutions can verify the trustworthiness of entire networking paths, rendering them unable to detect complex topology attacks that span multiple hops.

We advocate for a solution capable of verifying entire paths between endpoints, implemented as an SDN application, without requiring any modifications to the SDN controller. Some existing path validation solutions involve the collection of packet counts according to the relevant matching rules [15], [16]. Another approach requires adding a validation header to each packet to record the switch information along its transmission path [17]–[20]. However, these solutions perform per-switch operations on the CPU, leading to significant computational overhead, and generate high network overhead when handling large volumes of packets, which impacts both user traffic and switch-to-controller communication.

To tackle the above challenges, this paper introduces

PathSafe, a tool designed to verify available network paths between any pair of switches and determine the number of hops involved. The solution comprises a lightweight data plane component that runs on each switch, responsible for monitoring packets and marking probe packets at line speed, and a control plane component that collects probes and verifies the correctness of paths. To make it control plane agnostic, the solution is built on top of the SDN controller at the application level rather than modifying it directly. This approach mitigates the insecure and inefficient topology services currently available in the controller. We designed PathSafe to be further customized. For example, since the overhead is minimal, it can verify paths either periodically or on-demand. Similarly, PathSafe can verify all paths between two nodes, all nodes, or only a subset of paths. We designed the solution to be compatible with P4-enabled switches, a domain-specific programming language for specifying the behavior of network data planes [21], [22], where it was then implemented in a prototype. Results obtained over the Mininet emulator confirm the lightweight approach and implementation, which leads to minimal overhead and execution time compared to state-of-the-art solutions such as [17], [20]. The results also validate the scalability of PathSafe at the increasing of sizes and paths to verify.

The rest of the paper is structured as follows. We discuss similar and existing solutions in the context of network topology verification in Section II. We overview the PathSafe functionalities in Section III, and the security threats addressed in Section IV. We then evaluate the impact of PathSafe over the P4 testbed in Section VI and conclude the paper in Section VII.

II. Related Work

The efficacy and efficiency of network management and programmable data planes hinge on monitoring various network parameters [1], including bandwidth, latency, capacity, the number of hops, and the network topology. These metrics allow the controller to assess network behavior and enact real-time modifications to optimize performance. In fact, SDN protocols are not designed to validate packet paths, allowing compromised switches to divert packets from their intended paths, leaving the controller in the dark [7], [23].

The absence of robust authentication mechanisms for both the controller and LLDP packets is a fundamental vulnerability in SDN [24]. Extensive research efforts have been directed towards mitigating the highlighted security concerns [25], [26]. One approach consists of counting packets at the data plane level. In fact, as packets traverse each hop and undergo rule matching at each switch, the match count for a network flow should remain consistent across all hops. Solutions such as FADE [27], iFADE [15], and FOCES [16] install dedicated rules to collect flow statistics, then verified by the controller to detect any irregularities. These methods are sensitive to packet losses and network congestion, which factors can lead to inaccurate packet counts in SDN switches.

Another approach to path validation in SDN involves appending a validation header to data packets. SDNSec [17] and RuleOut [18] propose that switches embed their proofs (or forwarding rules in the case of RuleOut) into a validation header, which can then be sent to the controller for validation. A Verifying Rule Enforcement (REV) technique [19], [28] enhances path validation by encrypting switch identification into the validation header upon packet reception. Additionally, REV employs a compressive message authentication code to minimize the size of the validation header. L-PVS [20] uses the validation header but pre-computes the validation information to reduce data plane computations. However, a critical shortcoming of these studies is the incurred computation overhead due to the execution of per-switch operations that demand the CPU and the lack of a comprehensive solution capable of counteracting common attacks yet offering a viable data-plane resolution. Moreover, although header compression can reduce switch storage and bandwidth usage, it burdens the controller for more computations.

To the best of our knowledge, no existing solutions offer secure path verification with the level of effectiveness and flexibility provided by our P4-implemented solution. Moreover, most other solutions rely on specific controller implementations, which is not a limitation of our approach.

III. PathSafe Design

We envision PathSafe primarily as a tool for network operators to verify segments of an SDN network topology, providing verified and cryptographically secure information. In this section, we start outlining the assumptions and potential attacks we aim to defend against, and then we describe how we integrate security into network path verification.

A. Threat model

Attacker model. We consider an SDN-based network that includes an SDN controller with insecure topology services, making it vulnerable to topology attacks [6]–[8]. It is assumed that the SDN network is composed of programmable switches, meaning the switches can be fully reprogrammed using the P4 programming language. Adversaries can gain control of hosts and switches, but the SDN controller is considered to be trustworthy. This assumption is common, as little could be done if the SDN controller were compromised. Through the compromised switches, adversaries can attempt various types of topology attacks by redirecting traffic over unauthorized paths to facilitate eavesdropping, performing man-in-the-middle attacks, or bypassing security middleboxes [10], [11].

The goal of adversaries is to manipulate the tool's output, leading to the reporting of incorrect values to the user, whether it be the network operator or the controller. Such a scenario would adversely affect the network, as it would hinder the user's ability to make well-informed decisions, potentially compromising both the performance and security of the entire network infrastructure. The attacker's motivations could vary widely. One possible intent is to deny or disturb the network's service. Alternatively, the attacker may seek to manipulate network traffic to their advantage, such as by directing traffic through links that are under their surveillance, thereby enabling them to intercept and potentially extract sensitive information.

Assumptions. We assume that the controller and its connections to the switches are secure and trustworthy, with a Transport Layer Security (TLS) connection established between the controller and each switch, ensuring a secure communication channel for the control plane. This is the usual case in the SDN environments. Additionally, the controller possesses knowledge of the initial network topology, although this information remains unverified, lacking security assurances.

B. PathSafe Overview

For a more effective solution description, we refer to an example topology (shown in Fig. 1) where PathSafe could be used to verify all paths between switches S_1 and S_7. Solid blue lines represent data plane connections, and green dashed lines depict control plane connections. Port numbers for each switch are also indicated. This topology comprises seven switches, a single controller, and two hosts, providing a foundation for analyzing our proposed protocol solution and conducting a security assessment. The chosen topology encompasses the full range of characteristics typically found in a conventional network, including the presence of multiple loops, various paths between hosts, and direct paths devoid of loops.

Protocol overview. The SDN controller is responsible for creating the probe packets and initiating the protocol, either by defining the events that trigger its launch or by allowing network operators to collect information from the available network paths on demand. The protocol begins when the controller injects a control packet (i.e., *probe*) into the data plane, specifically targeting the initial node of interest. Our protocol is designed so that the probe navigates through the data plane, collecting information from each node in the path until it reaches the designated endpoint. The information added to the packet is cryptographically protected-preventing tampering by attackers-by fresh keys generated from a pre-installed seed using a Key Distribution Function (KDF) within the controller. The key distribution is securely performed during the manufacturing process, avoiding the computational burden of dynamic key exchanges. Subsequently, the final switch in the path forwards the probe back to the controller, which then verifies the results. It is important to note that while the controller plays a crucial role in initiating and concluding the protocol, its involvement is confined to these stages, with the bulk of the protocol's operations being executed solely within the data plane.

The proposed protocol consists of three main phases: (i) the initialization phase, (ii) the probe forwarding phase (or measurement phase), and (iii) the verification phase (or termination phase).

1) Initialization phase: Whenever PathSafe is run, the first step is for the SDN controller to establish communication with the two nodes between which the available paths will be measured. Then, the SDN controller creates a fresh probe packet directed towards the initial node in the path (S_1), encapsulating the following details:

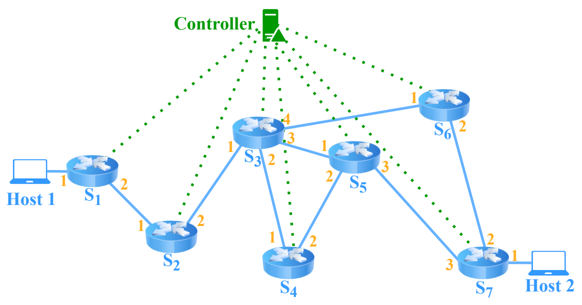

Fig. 1: Example topology featuring seven switches (S_1 to S_7), two hosts (*Host 1* and *Host 2*), and an *SDN controller*.

- **Session ID** (8 bits). Requesting path verification between two distinct points in the network and retrieving results constitutes one protocol session. Users can run PathSafe multiple times, and to distinguish between different runs, a unique identifier for each protocol session is necessary. This identifier facilitates differentiation among concurrent sessions and allows the programmable switches within the data plane to manage the control packet forwarding accurately. Therefore, the controller must randomly select a value between 0 and 255 to generate a new Session ID for each protocol run. Once chosen, a Session ID cannot be reused until the corresponding session concludes. Consequently, up to 256 concurrent sessions can be maintained.
- **TTL value** (8 bits). This value is decremented by each hop and allows the user to define the probe's lifespan within the data plane, potentially restricting interest to paths that do not exceed the TTL value. Additionally, this value assists the controller in determining the hop count metric.
- **Expiration Time** (4 bytes). This timestamp specifies when the probe packet is deemed obsolete, prompting the relevant switches to discard the probe afterwards.
- **VC** (72 bits). A variable component is initially appended by the first switch and later updated by others. It concatenates the egress port of the current hop and the newly computed MAC with the secret key. 8 bits are used to express the port, and 64 bits are used to store the MAC.

As indicated earlier, the probe will then traverse a series of programmable switches until it reaches the terminal node. During the initialization phase, the SDN controller also informs the destination node of the probe packet's arrival. To achieve this, the controller must dispatch a notification packet to the final switch in the path (S_7), where this notification packet comprises the following information:

- **Session ID** (8 bits). It matches the identifier allocated in the probe packet, signaling to the switch that it constitutes the final node within the path.
- **Wait Time** (4 bytes). Denotes the duration the terminal switch will anticipate the probe's arrival. Beyond this interval, the switch is instructed to discard any incoming control packets bearing the corresponding session ID.

These actions effectively mark the conclusion of the protocol's initiation phase.

2) Probe forwarding phase: After receiving the necessary data, the initial switch in the path can begin executing the protocol. The data plane probe forwarding process (excluding possible loops[1]) is given in Fig. 2. We can observe how the probe packets traverse the network to reach the destination switch S_7. The core principle is ensuring the integrity of the probe as it moves through the data plane, which involves protecting against unauthorized modification of the probe packets. To achieve this, each switch (S_i) that comes into contact with the probe will compute a Message Authentication Code (MAC) using its symmetric key K_i over the critical information contained within the probe. The switch then incorporates this MAC into the probe itself.

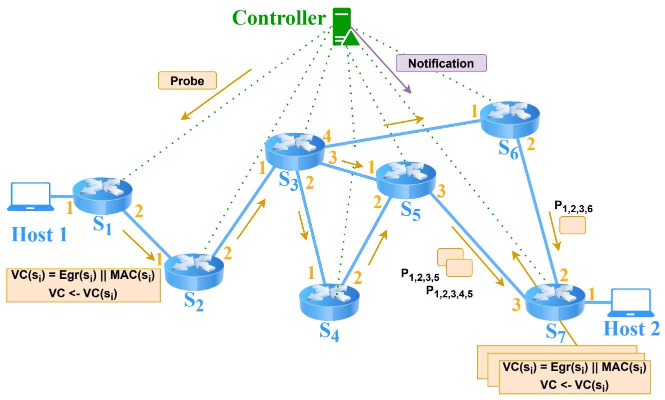

Fig. 2: Example of PathSafe's behavior when forwarding probe packets through the network (at the data plane).

Since each switch is required to perform basic symmetric key cryptographic operations, such as generating Message Authentication Codes (MACs), we designed the operations in PathSafe to be compatible with current switch architectures. As mentioned earlier, the advent of high-speed programmable switches allowed easy customization within the network, improving security, performance, and reliability, but posed limitations in possible actions of the data plane. Therefore, we designed hashing operations that can meet the requirements for running entirely in the data plane and at line rate while satisfying high security for short-input MACs (more details in Section V). More concretely, before passing the probe to the next hop, every switch in the path will carry out the procedures specified in (1).

$$\begin{aligned} &B = SessionID \parallel ExpTime \\ &MAC(S_i) = MAC_{K_i}(TTL_{S_i} \parallel egr(S_i) \parallel VC(S_p) \parallel B) \\ &VC(S_i) = egr(S_i) \parallel MAC(S_i) \end{aligned} \tag{1}$$

In (1), the terms and symbols are defined as follows:

- The B denotes the constant part of the probe packet, which remains unchanged for every switch along the path within a single protocol session. This part is referred to as the base of the probe packet.
- The $egr(S_i)$ denotes the identifier of the egress port through which switch S_i forwarded the probe. This information helps the controller confirm not only the connections between switches in the network but also the specific ports through which they are connected.
- The $VC(S_i)$ represents the variable components of the protocol associated with switch S_i.
- The S_p represents the previous switch for the corresponding path. It is important to note that there is no prior switch for the first switch, so the term $VC(S_p)$ is defined as $\{\emptyset\}$.
- The TTL_{S_i} indicates the probe packet's conventional Time-To-Live (TTL) value when it is dispatched by switch S_i.

The initial switch on the path attaches the $VC(S_i)$ term to the probe packet. Subsequent switches that receive the probe will update this term with their respective values, recalculating the MAC and updating the port number term accordingly. This process creates a "chain of MACs," where each new MAC is generated on the previous one. Thus, when a new switch in the path updates the MAC, this new MAC includes information from all preceding switches along the path. When a sequence of switches S_1 to S_n (where n is a natural number) amends the probe packet, we represent this as $P_{1,..,n}$.

Each switch is configured to multicast the probe packet to all its ports, excluding the port from which the probe originated, to prevent needless duplication or looping of the probe. It should be noted that this approach does not completely solve the issue of probe packets looping within the network but mitigates it. However, preventing endless loops can be achieved by utilizing the correct TTL values for the probe packets. A single switch may receive several probe packets from the same session, which is anticipated given a switch's potential involvement in numerous paths between two network nodes. As the protocol advances, the path's terminal node will begin accumulating probe packets from the relevant session. It will persist in collecting and storing these packets until the *Wait Time*, as specified by the controller at the onset of the protocol, elapses. Subsequently, the terminal node will dispatch all amassed probe packets to the control plane (the controller) for verification.

3) Verification phase: The final stage of the protocol involves the controller executing path verification. The controller can determine the relationship between the probes and their respective network paths by examining the port forwarding data within each received probe. Knowing the shared secret keys, base B, initial TTL, the correct path $S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_n$, it can computes its version of $VC^c(S_1)$ (controller variables are herein denoted with apex c)

$$VC^c(S_1) = egr(S_1)||MAC^c_{K_1}(TTL^c_{S1}||egr(S_1)||B), \tag{2}$$

and then it can calculate its version of the variable component for the last switch $VC^c(S_n)$ by iteratively calculating (1) for every switch on the designated path.

[1]This is a realistic scenario as probe $P_{1,2,3,4}$ can arrive to switch S_5 before $P_{1,2,3}$ and thus close the corresponding port for forwarding.

The controller compares its computed final $VC^c(S_n)$ with the $VC(S_n)$ embedded in the corresponding probe packet. A match confirms the integrity of the path in question. In contrast, a mismatch suggests that the controller's topology information might be out of date or that some switches in the network are potentially compromised. When one anomaly is detected, the controller is aware of a possible attack or misconfiguration, but it's not aware of which switch has been compromised. To this end, it can narrow down the verification to a subset of switches along $S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_n$ to detect the compromised link. This search can be performed with any known search algorithm. Ultimately, this verification process is replicated for all paths in the network or only for monitored ones. At the end of this process, the controller compiles a list of authenticated paths and associates to each hop count metric for possible future diagnostics.

Correctness Analysis: If the probe packet follows the designated path, the validation field $VC(S_n)$ (received from the data plane) should equal the validation field $VC^c(S_n)$ computed in the controller. The variable $VC(S_n)$, at the final hop n is computed in the following way:

$$\begin{aligned} VC(S_n) &= egr(S_n)||MAC(S_n) = \\ &= egr(S_n)||MAC_{K_n}(TTL_{Sn}||egr(S_n)||VC(S_{n-1})||B) \end{aligned} \tag{3}$$

where for S_1 the component is obtained as:

$$VC(S_1) = egr(S_1)||MAC_{K_1}(TTL_{S1}||egr(S_1)||B) \tag{4}$$

The controller, during the verification phase, possesses all the information needed to compute its version of $VC^c(S_i)$ for each hop in the designated path and obtains $VC^c(S_n)$. From (2) and (3), we observe that $VC^c(S_n) = VC(S_n)$ if and only if the probe packet has traversed designated path $S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_n$.

C. When to run PathSafe?

Given the workflow of PathSafe, we envision that our solution can be utilized in several ways:

- When the communication between two switches is deemed critical, if the network operator finds the preliminary topology information insecure, he can run *PathSafe* to verify all available paths or just a subset. Significant discrepancies between the *PathSafe* findings and the initial topology data are sufficient to restrict communications to certain paths deemed secure while avoiding others. The ability to process data at line speed is particularly crucial in this scenario, as it helps to reduce delays in making routing decisions.
- Up-to-date information on available network paths and hop counts are essential to make well-informed traffic routing and load balancing decisions [29]. Given the dynamic nature of networks, particularly in virtual SDN settings, the operator can employ the tool to update the controller's knowledge of the network's topology.
- Additionally, the tool can serve for debugging purposes. In case of performance drops in specific areas, verifying the available network paths can be instrumental in pinpointing the underlying issues. The high-speed and dynamic characteristics of SDN environments necessitate the capability to process information at line speed, ensuring that the controller has access to the most recent data for troubleshooting and performance optimization.
- The *traceroute* (or *tracert*) is a widely-used diagnostic tool in IP networks that enables network operators to trace the path from a source to a destination host [30]. Within SDN environments, network operators can utilize PathSafe for the same purposes. Thanks to its flexibility, it allows network operators to select a specific path or to verify all available paths between two points. This is unlike *traceroute*, which typically provides only a single path output. This feature is particularly beneficial for comprehensive network analysis and troubleshooting.

IV. Security analysis

In this section, we discuss the protocol's effectiveness in countering security threats while identifying any persisting vulnerabilities or risks. We analyze attack scenarios where an adversary gains control over one or more switches in the network to alter the protocol's output or deny access to its users, effectively resulting in a Denial of Service (DoS). We will examine four distinct attack scenarios, considering both solitary and multiple malicious switches, including an attack that requires at least two compromised switches.

Probe modification attack. In this attack scenario, the adversary aims to tamper with the probe packet to mislead the protocol user. However, such manipulation proves impractical because the probe packet's sensitive fields are safeguarded by a Message Authentication Code (MAC). Any field alteration would result in a discrepancy between the MAC values computed by the controller and those embedded in the probe packet. Consequently, the MAC safeguards the integrity of the probe packet's information, effectively thwarting any attempts at modification-based attacks. It is important to note that a 64-bit MAC is sufficient for our application, as one protocol run (session) usually lasts only a few seconds. Within this time frame, it is unfeasible for the adversary to create a valid MAC for a modified probe packet since the session number differs for each new protocol run, meaning the attacker would have to start again each time. This security mechanism effectively prevents manipulation regardless of the number of malicious switches.

Probe relay attack. In this strategy, the adversary attempts to deceive the controller by simply relaying the probe packet through a malicious switch without altering it. By doing so, the malicious switch would become invisible on the path from the controller's viewpoint. For example, consider a scenario where switch S_3 is compromised and forwards the probe $P_{1,2}$ through its port 4. From the controller's perspective, this could falsely suggest a direct connection between S_2 and S_6, as depicted in Fig. 3 with a red dashed line. However, discrepancies might arise from the analysis of other probe packets. Specifically, it could be observed that port 2 of S_2 is linked to both S_3 and S_6 simultaneously. Even if S_3 were to relay $P_{1,2}$

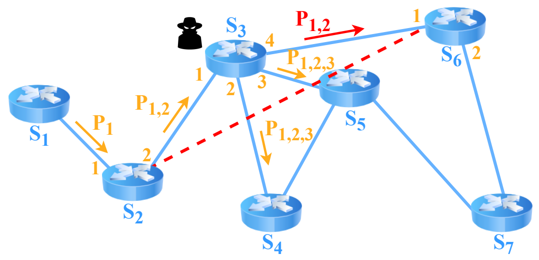

Fig. 3: Single switch relay attack example.

through all its ports, it would still result in the same port-wise inconsistencies. The critical takeaway is that a single physical port can only establish a connection to one switch at any moment.

To circumvent such inconsistencies, the adversary may opt to relay the probe packet through a specific port (e.g., port 4 in Fig. 3) while refraining from forwarding $P_{1,2,3}$ through ports 2 and 3. This action effectively suggests the non-existence of links S_3–S_4 and S_3–S_5. From the controller's perspective, this creates no discrepancies based on the probe packets received.

The presence of a malicious switch thus remains undetected by the controller until it exposes itself by forwarding traffic through different ports than those used for the probe packets, leading to observable discrepancies. Therefore, the impact of such an attack is relatively limited since the malicious switch's activities are restricted to merely relaying packets. This constraint significantly limits the attacker's ability to disrupt network traffic, as the switch must remain concealed, engaging solely in packet relaying. For example, should the attacker route regular traffic through an unverified link, such as from S_3 to S_5, this action would be unexpected by both the controller and S_5 (presuming the controller has informed each switch of legitimate links). The non-compromised S_5 would then detect packets arriving on an unverified port and could promptly notify the controller of this irregularity, thereby enabling the controller to identify the inconsistency.

In the scenario involving multiple malicious switches, if they do not collaborate, the attack scenario is treated as multiple individual attacks. Otherwise, the attacker can create a "relay chain," enabling the continuous relay of probe packets. As a result, the malicious switches have a significantly wider range within the network, allowing them to relay packets across longer distances. However, they are still restricted to relaying only one probe packet per switch, similar to the previous example. As discussed earlier, if any switch were to forward more than one probe, the controller could detect inconsistencies in port connections.

Denial-of-Service attack. In a Denial of Service (DoS) attack, the attacker's objective is to interrupt or hinder the service facilitated by the protocol. A malicious switch can launch a DoS attack by failing to forward any probe packets. The controller can detect such an attack by validating various paths and isolating the malicious switch. It can attempt to verify alternative routes that exclude the compromised switch from the network, thereby identifying it. It should be acknowledged that this method necessitates multiple iterations of the protocol and incurs additional time expenditure.

In the case of multiple malicious switches that refuse to forward probe packets, it becomes challenging for the controller to draw clear conclusions from the received probe packets and accurately identify the malicious switches. This leads to an increased number of protocol runs to identify the source of the problem.

Out-of-band attack. The out-of-band (OOB) attack scenario necessitates the collaboration of at least two malicious switches capable of communicating via an out-of-band channel. The basic idea is that the attacker controlling two compromised switches can establish a separate, new, out-of-band link using alternative communication methods. This alternative communication pathway might leverage technologies such as 5G or Wi-Fi networks. For this to be possible, both switches need to possess authentic yet unused ports—a plausible assumption given that certain programmable switches, such as the Intel Tofino 2, can support up to 256 ports [31]. In this scenario, the compromised switches follow our protocol and forward their corresponding probe packets to both normally utilized ports and ones used for an OOB link. The link created by the attacker acts as a genuine, real link in the network (assuming the performance of the alternative communication is reliable). Upon examination, the controller is misled into believing in the existence of a direct connection between the compromised switches. This attack modality is considerably more viable in a physical SDN setup, which is the focus of our analysis. It should also be noted that this type of attack is less likely to occur, demands advanced capabilities from the attacker, and is barely addressed by other studies, e.g., [17].

V. Protocol Implementation with P4-enabled Switches

In this section, we delve into the implementation details of our prototype over the P4 data plane.

Programming P4 switches. The advent of SDN brought flexibility and programmability to networks; however, traditional networking devices, such as fixed-function ASICs, lacked this same flexibility. In response, P4 [21] emerged in 2014 as a domain-specific language tailored for networking devices, designed to provide reconfigurability, protocol independence, and target independence.

A P4 program describes the whole workflow of packet-forwarding devices. Whenever a packet is received, custom headers are extracted by the Parser following predefined sequences based on header types. Then, it is passed to an ingress match-action pipeline, which determines the forwarding decision and sends the packet contents and metadata to the buffer. For each egress port, the packet is processed by an egress match-action pipeline, where, eventually, the recirculation of the packet can be established. After further processing, the packet is reassembled in the Deparser and, finally, forwarded.

BMv2 [32] is the most widely-used software platform for targeting P4 programs. It serves as a platform for developing and testing P4 data and control plane programs, offering extensive flexibility and enhanced debugging features. These advantages make it an ideal choice for developing and testing P4 programs.

Security functions in the data plane. The implementation of the security functions on the switches (see Section III-B) comprises SipHash, a set of keyed hash functions known as pseudorandom functions (PRFs), which create a hash from an input string using a secret key, yielding a result that appears random [33]. Because it is designed for high-speed execution, it efficiently fits the P4 constraints and is particularly efficient with short input strings. In fact, it is the only hash function that can be executed in a single pass through the Tofino pipeline.

The operation of SipHash can be summarized as follows: It processes an input string along with a 128-bit secret key, initializing four internal 64-bit state variables. The *SipHash-c-d* variant then executes c compression rounds followed by d finalization rounds on the input. These rounds, called SipRounds, are identical but include additional pre-processing and post-processing steps at certain points.

We integrated these cryptographic primitives into the P4 program *directly in the data plane*, without the need to run them in "extern" functions [34] or in the control plane. To compute the SipHash over the output port (as described in Eq. 1), which is not accessible in the Ingress pipeline of P4, we implemented the main SipHash computation in the Egress pipeline. When a packet is received, it passes through the Ingress pipeline, where its Multicast group is identified. This Multicast group determines the set of ports to which the packet will be forwarded, generating multiple copies that are then sent to the Egress pipeline. In the Egress pipeline, the hash computation is performed using the actual output port of each packet, performing the SipHash computation in one pipeline pass.

VI. PathSafe Evaluation

A. Experimental Settings

To experimentally assess the impact of PathSafe compared to other state-of-the-art, we run experiments over Mininet [35], a well-known network emulator, to create realistic networks[2]. In particular, we perform the experiments over *(i)* a small topology (Fig. 1) and *(ii)* one larger from the CAIDA Internet Topology Data Kit (ITDK) dataset [36] (AS 13576), which comprises 30 nodes with up to 90 concurrent paths for verification.

We compare our work against two state-of-the-art solutions:

- **SDNSec** [17]: for every packet in the network, each traversed switch computes two AES-CBC-MACs, one to update the path validation field and one to append as a Forwarding Entry for path enforcement. The overhead per packet is composed of a fixed size of 22 bytes and a variable cost of 8 bytes-per-switch.
- **LPV-S** [20]: it adds a fixed header of 16 bytes to every packet in the network. Each traversed switch updates the header fields, XOR-ing the old values with its key and its egress port key.

[2]The entire source code is available at https://github.com/dahara98/PathSafe_2024

In these solutions, each path is verified and the last switch is responsible for sending every header to the controller.

B. Overhead and execution time in different settings

In the first set of experiments, we assess the overhead in the network, defined as the maximum amount of $Kbps$ transmitted in a link during the execution of the protocol. For all experiments, we run 20 runs in each topology, and then we plot the average and the 90% confidence interval. We start considering the impact of the number of packets traveling the network during the execution of the protocols, reporting such overhead in Fig. 4a. For this experiment, we consider a path length of 7 (but similar results were observed for other path lengths). SDNSec introduces the most overhead, as 78 bytes are totally added to each packet. L-PVS's overhead is heavily reduced, as it only uses 16 bytes, but still linearly grows with the number of packets. On the contrary, with PathSafe, we can monitor paths on a frequent basis, and this strategy has the advantage of reducing overhead while still maintaining security. Moreover, PathSafe sends one probe and one notification packet with a fixed size to verify multiple paths, resulting in a fixed overhead of 0.4 Kbps.

In Fig. 4b, we represent how the path length impacts the overhead during a 10-packet flow between two hosts (to reflect other benchmarks settings [17], [20]). In SDNSec, each switch appends an 8-byte Forwarding Entry to every packet, incrementing the overhead with each hop. In contrast, L-PVS and PathSafe do not experience this incremental overhead, as they use a fixed byte amount. However, our solution can minimize the overhead by eliminating the need to verify each flow packet. We then assess the overhead based on the number of verified paths (Fig. 4c). Each path is identified by a different flow constituted by 100 packets. Since PathSafe sends probe packets via multicast, it can verify multiple paths in a single execution of the protocol, thereby once again proving to be the least impactful protocol.

In the second set of experiments, we assess the end-to-end execution time in the same settings as before, but considering only one packet per flow to better assess the computational overhead. The end-to-end execution time is measured as the time since the first packet enters the data plane until the end of the verification process at the control plane. Since L-PVS and SDNSec need CPU to compute the validation field, our solution achieves the lowest execution time as we perform line-speed operations in the switch (Fig 5a). Additionally, since the benchmarks verify each packet, the time required for the execution linearly grows with the number of packets sent over the network, while PathSafe takes about 80 ms on average. cClearly, the execution time may differ when transitioning from an emulator to a hardware-based device. Nonetheless, our objective here is to evaluate the performance of different solutions under equal conditions, and the results unequivocally indicate that our solution can efficiently scale with larger networks.

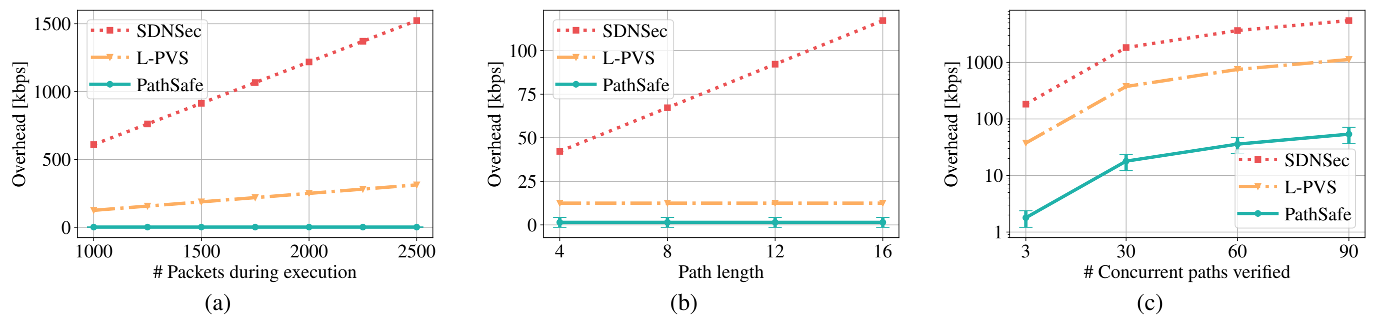

Fig. 4: Overhead introduced by solutions when (a) varying the number of packets, (b) varying the length of the verified path, (c) varying the number of verified paths. PathSafe adds a fixed or minimum overhead.

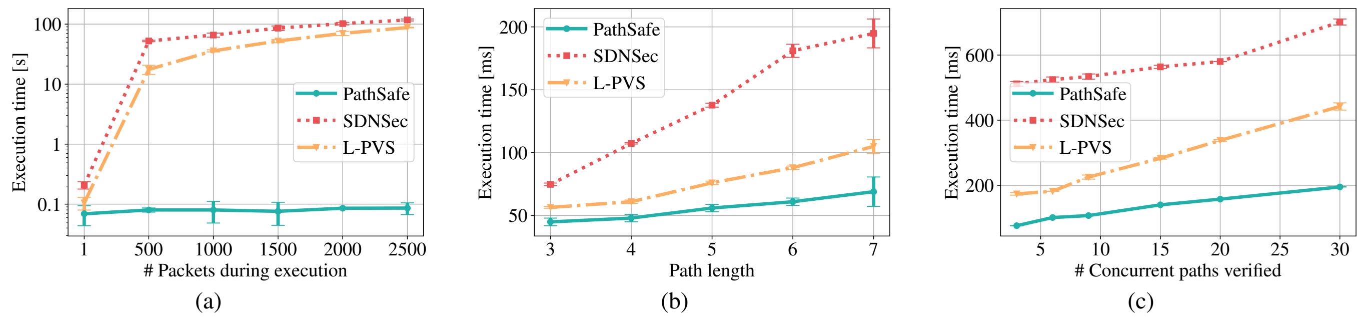

Fig. 5: Execution time of solutions when (a) varying the number of packets, (b) varying the length of the verified path, (c) varying the number of verified paths. PathSafe adds a fixed or minimum overhead.

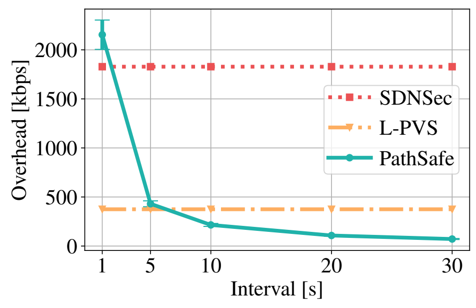

Fig. 6: Overhead introduced when varying the interval between two executions of PathSafe.

As far as concerns the dependency of the execution time on the path length (Fig 5b), the effects of a longer transmission time and computation due to the longer path are negligible in our solution, which employs 77 ms on average. This is not true for the other solutions that experience an increased delay due to more CPU computations. Moreover, Fig 5c shows the time taken when multiple concurrent paths are verified. Once again, SDNSec employs the most time due to the complex operations at the CPU. Meanwhile, L-PVS performs lightweight per-flow operations and computes the hash directly for the assigned route, making the verification process faster. On the other hand, our solution shows great scalability by being able to verify many paths within a single execution, e.g., employing about 200 ms to verify 30 paths concurrently.

Finally, as emerged from previous graphs, the interval used in PathSafe in securing paths has a significant impact on the performance and the security, as the detection of various kinds of attacks (e.g., probe modification, probe relay, and DoS attacks) depends on this parameter. However, since results indicate that the execution time is negligible, the detection time is mainly (and only) affected by this verification frequency. Remarking that PathSafe can be executed on-demand (or for predefined events) or periodically, we now discuss possible insights for choosing this verification interval.

Fig. 6 shows the overhead introduced by different time intervals for the verification (a low interval implies a more reactive solution). A period of 5 s results in about 7 s to detect the attacks, adding a comparable overhead w.r.t. L-PVS. On the contrary, running PathSafe every 10 s results in about 12 s of detection time but less overhead. In conclusion, we argue that running PathSafe every 5 to 10 s results in a good trade-off between security and performance, achieving less overhead and faster verification time w.r.t. SDNSec, and a comparable performance w.r.t. L-PVS. Thanks to its high flexibility, network operators can use PathSafe while maintaining control over the bandwidth overhead.

VII. Conclusion

In this paper, we introduced the *PathSafe* tool, designed to address security concerns in SDN by securely verifying path information within the network. By leveraging Message Authentication Codes (MACs), the protocol effectively ensures the integrity of path information in different scenarios, thereby mitigating security threats. Additionally, we presented a proof of concept implementation using the P4 programming language. Experiments on Mininet validated the efficacy of PathSafe's design and implementation compared to two benchmarks in terms of execution time and overhead introduced.

Acknowledgment

This work was supported by CyberSecurity Research Flanders with reference number VR20192203, by the European Union's Horizon Europe project under grant agreement No 101070516 (NebulOuS) and by the Spanish Ministry of Economic Affairs and Digital Transformation and the European Union NextGenerationEU (grant TSI-063000-2021- 44).

References

[1] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 103, pp. 14–76, 2015.

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

[3] S. W. Shin, P. Porras, V. Yegneswara, M. Fong, G. Gu, and M. Tyson, "FRESCO: Modular Composable Security Services for Software-Defined Networks," in *20th Annual Network & Distributed System Security Symposium (NDSS)*. Usenix, 2013.

[4] A. Sacco, F. Esposito, and G. Marchetto, "Rope: An architecture for adaptive data-driven routing prediction at the edge," *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 986–999, 2020.

[5] A. Abdou, P. C. van Oorschot, and T. Wan, "Comparative analysis of control plane security of sdn and conventional networks," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, pp. 3542–3559, 2018.

[6] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures." in *NDSS*, vol. 15, 2015, pp. 8–11.

[7] R. Skowyra, L. Xu, G. Gu, V. Dedhia, T. Hobson, H. Okhravi, and J. Landry, "Effective topology tampering attacks and defenses in software-defined networks," in *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 374–385.

[8] E. Marin, N. Bucciol, and M. Conti, "An in-depth look into sdn topology discovery mechanisms: Novel attacks and practical countermeasures," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. Association for Computing Machinery, 2019, p. 1101–1114.

[9] S. Khan, A. Gani, A. W. Abdul Wahab, M. Guizani, and M. K. Khan, "Topology Discovery in Software Defined Networks: Threats, Taxonomy, and State-of-the-Art," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 1, pp. 303–324, 2017.

[10] P.-W. Chi, C.-T. Kuo, J.-W. Guo, and C.-L. Lei, "How to detect a compromised sdn switch," in *Proceedings of the 1st IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2015, pp. 1–6.

[11] H. A. Noman and O. M. Abu-Sharkh, "Code injection attacks in wireless-based internet of things (iot): A comprehensive review and practical implementations," *Sensors*, vol. 23, no. 13, p. 6067, 2023.

[12] T. Alharbi, M. Portmann, and F. Pakzad, "The (in)security of Topology Discovery in Software Defined Networks," in *IEEE 40th Conference on Local Computer Networks (LCN)*. IEEE, 2015, pp. 502–505.

[13] A. Azzouni, R. Boutaba, N. T. M. Trang, and G. Pujolle, "sOFTDP: Secure and Efficient OpenFlow Topology Discovery Protocol," in *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2018, pp. 1–7.

[14] F. Pakzad, M. Portmann, W. L. Tan, and J. Indulska, "Efficient topology discovery in OpenFlow-based software defined networks," *Computer Communications*, vol. 77, pp. 52–61, 2016.

[15] Q. Li, Y. Liu, Z. Liu, P. Zhang, and C. Pang, "Efficient forwarding anomaly detection in software-defined networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 11, pp. 2676–2690, 2021.

[16] P. Zhang, F. Zhang, S. Xu, Z. Yang, H. Li, Q. Li, H. Wang, C. Shen, and C. Hu, "Network-wide forwarding anomaly detection and localization in software defined networks," *IEEE/ACM transactions on networking*, vol. 29, no. 1, pp. 332–345, 2020.

[17] T. Sasaki, C. Pappas, T. Lee, T. Hoefler, and A. Perrig, "SDNsec: Forwarding accountability for the SDN data plane," in *2016 25th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2016, pp. 1–10.

[18] S. Xi, K. Bu, W. Mao, X. Zhang, K. Ren, and X. Ren, "RuleOut Forwarding Anomalies for SDN," *IEEE/ACM Transactions on Networking*, vol. 31, no. 1, pp. 395–407, 2022.

[19] P. Zhang, H. Wu, D. Zhang, and Q. Li, "Verifying Rule Enforcement in Software Defined Networks With REV," *IEEE/ACM Transactions on Networking*, vol. 28, no. 2, pp. 917–929, 2020.

[20] B. Hu, Y. Bi, K. Wu, R. Fu, and Z. Huang, "A lightweight path validation scheme in software-defined networks," in *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 2024, pp. 1–10.

[21] 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.

[22] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, "A survey on data plane programming with P4: Fundamentals, advances, and applied research," *Journal of Network and Computer Applications*, vol. 212, p. 103561, 2023.

[23] K. Agarwal, E. Rozner, C. Dixon, and J. Carter, "SDN traceroute: Tracing SDN forwarding without changing network behavior," in *Proceedings of the third workshop on Hot topics in software defined networking*, 2014, pp. 145–150.

[24] T. Alharbi, M. Portmann, and F. Pakzad, "The (in) security of topology discovery in software defined networks," in *IEEE 40th Conference on Local Computer Networks (LCN)*. IEEE, 2015, pp. 502–505.

[25] J. Kim, E. Marin, M. Conti, and S. Shin, "EqualNet: a secure and practical defense for long-term network topology obfuscation," in *29th Annual Network and Distributed System Security Symposium*, 2022.

[26] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "Sphinx: detecting security attacks in software-defined networks," in *NDSS*, vol. 15, 2015.

[27] C. Pang, Y. Jiang, and Q. Li, "FADE: Detecting forwarding anomaly in software-defined networks," in *IEEE International Conference on Communications (ICC)*. IEEE, 2016, pp. 1–6.

[28] P. Zhang, "Towards rule enforcement verification for software defined networks," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[29] J. W. Guck, A. Van Bemten, M. Reisslein, and W. Kellerer, "Unicast QoS Routing Algorithms for SDN: A Comprehensive Survey and Performance Evaluation," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 388–415, 2018.

[30] die.net, "traceroute(8) - Linux man page," URL: https://linux.die.net/man/8/traceroute, accessed: 24/04/2023.

[31] Intel, "Intel Tofino 2," URL: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html, accessed: 15/04/2023.

[32] P4, "p4language," URL: https://github.com/p4lang, accessed: 02/01/2023.

[33] J.-P. Aumasson and D. J. Bernstein, "Siphash: a fast short-input prf," in *International Conference on Cryptology in India*. Springer, 2012, pp. 489–508.

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

[35] R. L. S. de Oliveira *et al.*, "Using Mininet For Emulation And Prototyping Software-Defined Networks," in *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)*, 2014, pp. 1–6.

[36] CAIDA, "The CAIDA Internet Topology Data Kit (ITDK)," URL: https://www.caida.org/catalog/datasets/internet-topology-data-kit/, accessed: 12/06/2024.