An Implementation of Route Probing and Validation on PolKA

Henrique Coutinho Layber†, Roberta Lima Gomes†, Magnos Martinello†, Vitor B. Bonella†, Everson S. Borges‡, Rafael Guimarães†‡

†Department of Informatics, Federal University of Espírito Santo
‡Department of Informatics, Federal Institute of Education Science and Technology of Espírito Santo

Abstract

This paper presents an implementation of a route probing and validation mechanism for the PolKA protocol. The mechanism is based on a composition of checksum functions on stateless core switches, which allows a trusted party to verify if a packet traversed the network along the path defined by the source.

Keywords

Verifiable Routing; Path Verification; Proof-of-transit; In-networking Programming

1. Introduction

Ever since Source Routing (SR) was proposed, there has been a need to ensure that packets traverse the network along the paths selected by the source, not only for security reasons but also to ensure that the network is functioning correctly and correctly configured. This is particularly important in the context of Software-Defined Networking (SDN), where the control plane can select paths based on a variety of criteria.

In this paper, we propose a new P4[1] implementation for a new protocol layer for PolKA[2], able to do validate the actual route used for a packet, and is available on GitHub[1]. This is achieved by using a composition of checksum functions on stateless core switches. It can then be checked by a trusted party that knows the `node_id`s independently.

2. Problem Definition

Let i be the source node (**i**ngress node) and e be the destination node (**e**gress node). Let path $\underset{i \to e}{P}$ be a sequence of nodes:

$$\underset{i \to e}{P} = (i, s_1, s_2, ..., s_{n-1}, s_n, e)$$

where

P Path from i to e.

s_n n-th core switch in the path.

i Ingress edge (source).

e Egress edge (destination).

In PolKA, the route up to the protocol boundary (usually, the SDN border) is defined in i [3]. i sets the packet header with enough information for each core node to calculate the next hop. Calculating each hop is done using Chinese Remainder Theorem (CRT) and the Residue

[1] https://github.com/Henriquelay/PolKA-halfsiphash/tree/remake/mininet/PolKA-example

Number System (RNS)[4], and is out of the scope of this paper. All paths are assumed to be both valid and all information correct unless stated otherwise.

The main problem we are trying to solve is path validation, that is, to have a way to ensure if the packets are actually following the path defined. Notably, it does not require verification, that is, listing the switches traversed is not required.

A solution should be able to identify if:

1. The packet has passed through the switches in the Path.
2. The packet has passed through the correct order of switches.
3. The packet has not passed through any switch that is not in the Path.

More formally, given a sequence of switches $\underset{i \to e}{P}$, and a captured sequence of switches actually traversed P_j, a solution should identify if $\underset{i \to e}{P} = P_j$.

3. Solution Proposal

Each node's execution plan is stateless and can alter the header of the packet, which we will use to detect if the path taken is correct. So, a node s_i can be viewed as a function $g_{s_i}(x)$.

In order to represent all nodes by the same function (for implementation purposes), we assign a distinct value k for each s node, and use a bivariate function $f\left(k_{s_i}, x\right) = f_{s_i}(x)$. By using functions in two variables, we force one of the variables to have any uniquely per-node value, ensuring that the function is unique for each switch, that is, $f_{s_y}(x) \neq f_{s_z}(x) \Leftrightarrow y \neq z$.

Using function composition is a good way to propagate errors since it preserves the order-sensitive property of the path, since $f \circ g \neq g \circ f$ in a general case. Each node will execute a single function of this composition, using the previous node's output as input. In this way:

$$\left(f_{s_1} \circ f_{s_2} \circ f_{s_3}\right)(x) = f\left(k_{s_3}, f\left(k_{s_2}, f\left(k_{s_1}, x\right)\right)\right)$$

s_i i-th switch in the path.

$f_{s_i}(x)$ Function representing switch s_i.

k_{s_i} Unique identifier for switch s_i.

3.1. Assumptions

1. Each node is assumed to be secure, that is, no node will alter the packet in any way that is not expected. This is a common assumption in SDN networks, where a trusted party is the only entity that can alter the network state.
2. PolKA and the proposed extension are open source, so it is assumed that any attacker can replicate a node perfectly.
3. Protocol boundary is IPv4. This means that PolKA is only used inside this network.

3.2. Setup

All implementation and experiments took place on a VM[2] setup with Mininet-wifi[5], and were targeting Mininet's[6] Behavioral Model version 2 (BMv2)[7]. Wireshark[8] was used to analyze packets, and Scapy[9] was used to parse packets programatically and automatic tests.

3.3. Implementation

By making the function f a checksum function, and the unique identifier k_{s_i} as the `node_id`, we apply an input data into a chain checksum functions and verify if they match. For additional

[2] Available on PolKA's repository https://github.com/nerds-ufes/PolKA

validation, we also integrate the calculated exit port into the checksum, covering some other forms of attacks or errors.

It was implemented as a version on PolKA, this means it uses the same `ethertype` `0x1234` and is interoperable with PolKA. Up-to-date PolKA headers were used (and upgraded from the forked version) to ensure compatibility. It uses the `version` header field to differentiate between regular PolKA version packets and what we call *probe* packets. PolKA packets uses version `0x01`, and probe packets uses version `0xF1`.

Figure 1 shows the used topology used in the experiments.

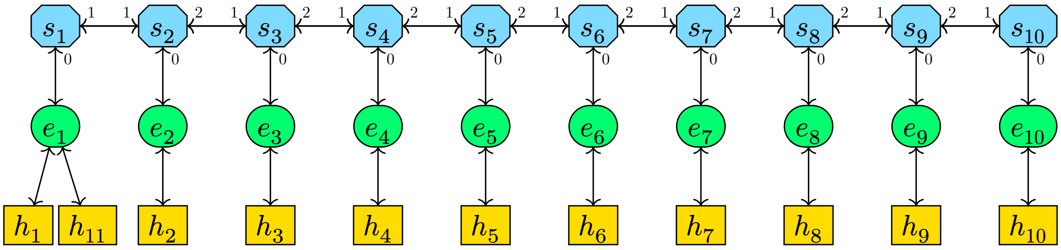

Figure 1: Topology setup.

s_n are core switches, e_n are edge switches, h_n are hosts.

This reads as follows: s_1 connects on port 1 to s_2's port 1, and on port 0 to e_1. s_2 connects on port 1 to s_1's port 1, and on port 2 to s_3's port 1, and to e_2 on port 0, and so on.

3.3.1. Parsing

Parsing is done in edge nodes as follows:

- If an IPv4 protocol `ethertype` field is detected (`0x0800`), it must be a packet from outside the network, it must be wrapped and routed by the same edge node that parsed it. Let call this process be called *encapsulation*;
- If a PolKA protocol `ethertype` field is detected (`0x1234`), it must be a packet from inside the network, since the protocol boundary is IPv4, the original IPv4 packet must be unwrapped. Let this process be called *decapsulation*.

On core nodes, the packet is only parsed as PolKA packets, but it can be either a regular PolKA packet or a probe packet. If a probe packet version is detected, the probe packet header is parsed aswell, otherwise the packet is treated a regular PolKA packet.

3.3.2. Encapsulation

PolKA headers consists of the route polynomial (`routeid`), along with `version`, `ttl` and `proto` (stores the original `ethertype`). `route_id` calculation is out of the scope of this paper.

A new header is added for probe packets, containing a 32 bit `key` and 32 bit `l_hash`.

During encapsulation of a probe packet, a random number is generated, and is used as `key`, for reproducibility and a seed for our composition. Edge nodes does not execute checksum functions and only repeats the key into the checksum field `l_hash`, so `node_id` for encapsulation is not required.

After encapsulation, the packet is sent to the next hop.

3.3.3. Composition

Every core node does checksum trying to congregate the previous `l_hash`, the calculated next hop port and it's own `node_id` into the 32 bit field. Currently, it is implemented as such:

$$\texttt{l\_hash} \leftarrow \text{CRC}_{32}(\text{exit port} \oplus \texttt{l\_hash} \oplus \texttt{node\_id})$$

The CRC_{32} checksum function used currently is the one available by BMv2 standard library, and through testing, it was found out to be ISO HDLC[3].

The algorithm was verified externally through another program simulating all the composition steps, with source available[4], making use of the `crc` library crate[5], and checked with gathered data.

3.3.4. Decapsulation

At the egress node, PolKA headers are dropped and the packet becomes an identical packet to what the ingress node received. The probe packet header is also dropped. The packet is then sent to the host. No checksum is calculated, so `node_id` is not required. Thus, together with Section 3.3.2, the `node_id` is not used for validation on edge nodes.

3.4. Example

A simple example of a packet traversing a network with 10 core switches is shown in the figure below. Exit port is calculated by PolKA.

$$\underset{e_1 \to e_{10}}{P} = (e_1, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, e_{10})$$

Node	`node_id`	`exit_port`	Calculation	`l_hash`
e_1		1	Generation	0x61e8d6e7
s_1	0x002b	1	$\text{CRC}_{32}(\texttt{0x61e8d6e7} \oplus \texttt{0x1} \oplus \texttt{0x2b})$	0xae91434c
s_2	0x002d	2	$\text{CRC}_{32}(\texttt{0xae91434c} \oplus \texttt{0x2} \oplus \texttt{0x2d})$	0x08c97f5f
s_3	0x0039	2	$\text{CRC}_{32}(\texttt{0x8c97f5f} \oplus \texttt{0x2} \oplus \texttt{0x39})$	0xeff1aad2
s_4	0x003f	2	$\text{CRC}_{32}(\texttt{0xeff1aad2} \oplus \texttt{0x2} \oplus \texttt{0x3f})$	0x08040c89
s_5	0x0047	2	$\text{CRC}_{32}(\texttt{0x8040c89} \oplus \texttt{0x2} \oplus \texttt{0x47})$	0xaa99ae2e
s_6	0x0053	2	$\text{CRC}_{32}(\texttt{0xaa99ae2e} \oplus \texttt{0x2} \oplus \texttt{0x53})$	0x7669685e
s_7	0x008d	2	$\text{CRC}_{32}(\texttt{0x7669685e} \oplus \texttt{0x2} \oplus \texttt{0x8d})$	0x03e1e388
s_8	0x00bd	2	$\text{CRC}_{32}(\texttt{0x3e1e388} \oplus \texttt{0x2} \oplus \texttt{0xbd})$	0x2138ffd3
s_9	0x00d7	2	$\text{CRC}_{32}(\texttt{0x2138ffd3} \oplus \texttt{0x2} \oplus \texttt{0xd7})$	0x1ef2cbbe
s_{10}	0x00f5	0	$\text{CRC}_{32}(\texttt{0x1ef2cbbe} \oplus \texttt{0x0} \oplus \texttt{0xf5})$	0x99c5fe05
e_{10}		0	Decapsulation	0x99c5fe05

Table 1: Packet trace while traversing $\underset{e_1 \to e_{10}}{P}$.

3.5. Adversity scenarios

3.5.1. Addition

An attacker PolKA switch s_{555} was added between s_5 and s_6, as shown in Figure 2. The packet was sent from e_1 to e_{10}. Suppose the attacking switch is properly connected in the ports that PolKA uses for this route, the packet is properly routed with PolKA, but the checksums will not match when validating the path in the future. The packet trace is shown in Table 2. Note the error propagating nature of composing checksums.

[3] https://reveng.sourceforge.io/crc-catalogue/all.htm#crc.cat.crc-32-iso-hdlc

[4] https://github.com/Henriquelay/PolKA_probe_checker/

[5] https://github.com/mrhooray/crc-rs

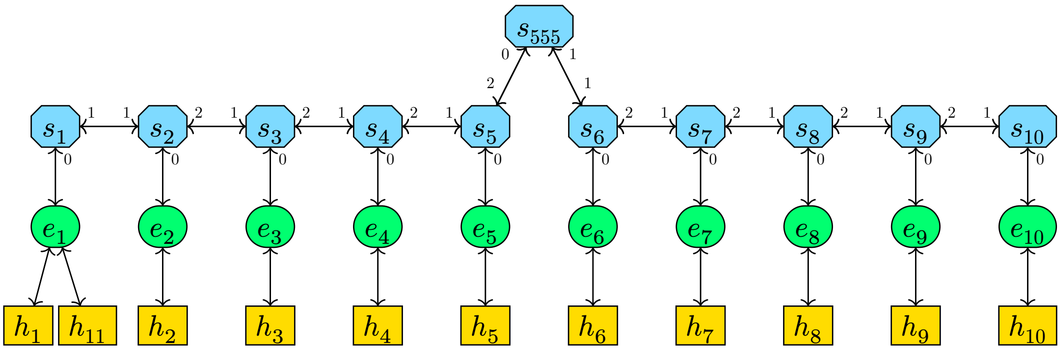

Figure 2: Topology setup for addition scenario.

Node	`node_id`	`exit_port`	Calculation	`l_hash`	Expected
e_1		`1`	Generation	`0xabadcafe`	`0xabadcafe`
s_1	`0x002b`	`1`	$\mathrm{CRC}_{32}(\texttt{0xabadcafe} \oplus \texttt{0x1} \oplus \texttt{0x2b})$	`0x432cf798`	`0x432cf798`
s_2	`0x002d`	`2`	$\mathrm{CRC}_{32}(\texttt{0x432cf798} \oplus \texttt{0x2} \oplus \texttt{0x2d})$	`0xe04df688`	`0xe04df688`
s_3	`0x0039`	`2`	$\mathrm{CRC}_{32}(\texttt{0xe04df688} \oplus \texttt{0x2} \oplus \texttt{0x39})$	`0xe8f0142c`	`0xe8f0142c`
s_4	`0x003f`	`2`	$\mathrm{CRC}_{32}(\texttt{0xe8f0142c} \oplus \texttt{0x2} \oplus \texttt{0x3f})$	`0xb452022a`	`0xb452022a`
s_5	`0x0047`	`2`	$\mathrm{CRC}_{32}(\texttt{0xb452022a} \oplus \texttt{0x2} \oplus \texttt{0x47})$	`0x4450d2d2`	`0x4450d2d2`
s_{555}	`0x0047`	`1`	$\mathrm{CRC}_{32}(\texttt{0x4450d2d2} \oplus \texttt{0x1} \oplus \texttt{0x47})$	`0x5b0fce3e`	
s_6	`0x0053`	`2`	$\mathrm{CRC}_{32}(\texttt{0x5b0fce3e} \oplus \texttt{0x2} \oplus \texttt{0x53})$	`0xc967a61d`	`0xe9367b57`
s_7	`0x008d`	`2`	$\mathrm{CRC}_{32}(\texttt{0xc967a61d} \oplus \texttt{0x2} \oplus \texttt{0x8d})$	`0xf6c27aa4`	`0x991182c1`
s_8	`0x00bd`	`2`	$\mathrm{CRC}_{32}(\texttt{0xf6c27aa4} \oplus \texttt{0x2} \oplus \texttt{0xbd})$	`0x38d0bc4f`	`0x35e72e11`
s_9	`0x00d7`	`2`	$\mathrm{CRC}_{32}(\texttt{0x38d0bc4f} \oplus \texttt{0x2} \oplus \texttt{0xd7})$	`0xb6ff911a`	`0xaa152eb9`
s_{10}	`0x00f5`	`0`	$\mathrm{CRC}_{32}(\texttt{0xb6ff911a} \oplus \texttt{0x0} \oplus \texttt{0xf5})$	`0x882d8e93`	`0x1a1573e7`
e_{10}		`0`	Decapsulation	`0x882d8e93`	`0x1a1573e7`

Table 2: Packet trace while traversing $\underset{e_1 \to e_{10}}{P}$ with an unexpected addition s_{555}.

3.5.2. Detour

An attacker tries to make a detour in the network, as shown in Figure 3. The packet was sent from e_1 to e_{10}, and passed through the detour, as shown in Table 3. The checksum will fail to validate when checking in the futures, as the function composition $f_{s_{555}}(x) \neq f_{s_6}(x)$. Note that this is only true when $k_{s_{555}} \neq k_{s_6}$. As stated, the key must be unique per node, and so must be kept secret.

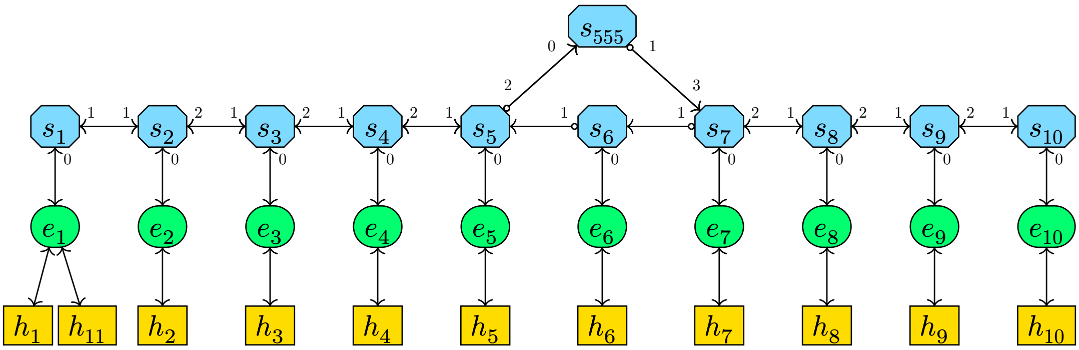

Figure 3: Topology setup for detour scenario.

Node	node_id	exit_port	Calculation	l_hash	Expected
e_1		1	Generation	0xbaddc0de	0xbaddc0de
s_1	0x002b	1	CRC$_{32}$(0xbaddc0de $\oplus$ 0x1 $\oplus$ 0x2b)	0x3ef96770	0x3ef96770
s_2	0x002d	2	CRC$_{32}$(0x3ef96770 $\oplus$ 0x2 $\oplus$ 0x2d)	0x2dca9942	0x2dca9942
s_3	0x0039	2	CRC$_{32}$(0x2dca9942 $\oplus$ 0x2 $\oplus$ 0x39)	0x11797334	0x11797334
s_4	0x003f	2	CRC$_{32}$(0x11797334 $\oplus$ 0x2 $\oplus$ 0x3f)	0x98081e3e	0x98081e3e
s_5	0x0047	2	CRC$_{32}$(0x98081e3e $\oplus$ 0x2 $\oplus$ 0x47)	0x3332e012	0x3332e012
s_{555}	0x0047	1	CRC$_{32}$(0x3332e012 $\oplus$ 0x1 $\oplus$ 0x47)	0x90a0df94	0x22996afd
s_7	0x0053	2	CRC$_{32}$(0x90a0df94 $\oplus$ 0x2 $\oplus$ 0x53)	0xbebe4372	0x8fa3987d
s_8	0x008d	2	CRC$_{32}$(0xbebe4372 $\oplus$ 0x2 $\oplus$ 0x8d)	0x5aafa7f2	0xf4b50950
s_9	0x00bd	2	CRC$_{32}$(0x5aafa7f2 $\oplus$ 0x2 $\oplus$ 0xbd)	0x649b8554	0xd0c29e67
s_{10}	0x00d7	0	CRC$_{32}$(0x649b8554 $\oplus$ 0x2 $\oplus$ 0xd7)	0xf46427bf	0x13ff41c1
e_{10}		0	Decapsulation	0xf46427bf	0x13ff41c1

Table 3: Packet trace while traversing $\underset{e_1 \to e_{10}}{P}$ with an unexpected detour.

3.5.3. Skipping

Miconfigured links can cause packets to skip nodes, as shown in Figure 4. The packet was sent from e_1 to e_{10}, and passed through the skipping, as shown in Table 4. The checksum will not match when validating the path in the future, as the packet did not pass through the expected nodes.

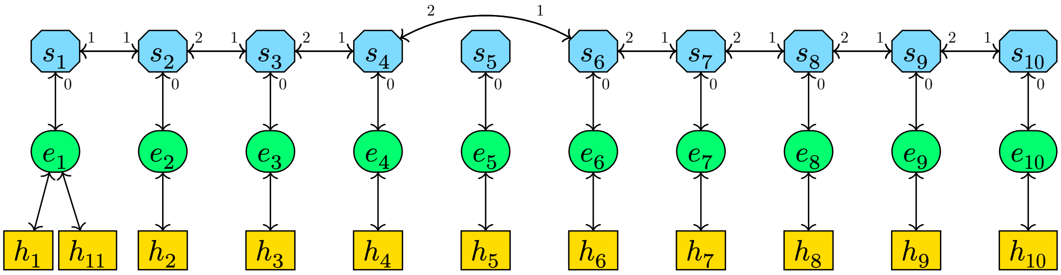

Figure 4: Topology setup for skipping scenario.

Node	node_id	exit_port	Calculation	l_hash	Expected
e_1		1	Generation	0x61e8d6e7	0x61e8d6e7
s_1	0x002b	1	CRC$_{32}$(0x61e8d6e7 $\oplus$ 0x1 $\oplus$ 0x2b)	0xae91434c	0xae91434c
s_2	0x002d	2	CRC$_{32}$(0xae91434c $\oplus$ 0x2 $\oplus$ 0x2d)	0x08c97f5f	0x08c97f5f
s_3	0x0039	2	CRC$_{32}$(0x8c97f5f $\oplus$ 0x2 $\oplus$ 0x39)	0xeff1aad2	0xeff1aad2
s_4	0x003f	2	CRC$_{32}$(0xeff1aad2 $\oplus$ 0x2 $\oplus$ 0x3f)	0x08040c89	0x08040c89
s_6	0x0053	2	CRC$_{32}$(0x8040c89 $\oplus$ 0x2 $\oplus$ 0x53)	0xb0437a53	0xaa99ae2e
s_7	0x008d	2	CRC$_{32}$(0xb0437a53 $\oplus$ 0x2 $\oplus$ 0x8d)	0x63589d0a	0x7669685e
s_8	0x00bd	2	CRC$_{32}$(0x63589d0a $\oplus$ 0x2 $\oplus$ 0xbd)	0x629b7b3b	0x03e1e388
s_9	0x00d7	2	CRC$_{32}$(0x629b7b3b $\oplus$ 0x2 $\oplus$ 0xd7)	0xbd53e851	0x2138ffd3
s_{10}	0x00f5	0	CRC$_{32}$(0xbd53e851 $\oplus$ 0x0 $\oplus$ 0xf5)	0x90bdf731	0x1ef2cbbe
e_{10}		0	Decapsulation	0x90bdf731	0x1ef2cbbe

Table 4: Packet trace while traversing $\underset{e_1 \to e_{10}}{P}$ with an unexpected skip.

3.6. Limitations

Upon developing the solution, a set of limitations were identified:

1. As with most cryptographic solutions, the system is only as secure as the key used. If the key is compromised, the entire system is compromised, since a malicious actor can easily generate the same checksums.
2. Replay attack is undetectable if metadata is disconsidered. This is due to the entry port not being included in the validation, which allows an attacker to replay the packet from a different port.

4. Future Work

The plan, as per the repository name implies, is to implement a non-reversible hash function, SipHash, more specifically, HalfSipHash[10], to be used instead of the CRC_{32}. This would make the system more secure, since CRC_{32} is a well-known as a checksum function that can be easily reversed[11]. Also, a proper data compression method for adding exit port and `node_id` into the checksum field is needed, since the current method is not optimal due to data loss.

In the future, it should into PathSec[12], and to do so the ingress edge needs to report the generated `key`, and the egress edge will report the final checksum directly to a blockchain, for auditability and accessibility. Having it directly report to a blockchain instead of a third party circumvents trust issues.

An interesting work can be done to use some sort of rotating key architecture to detect replay attacks. This is a hard problem, since the key must be rotated in a way that the attacker cannot predict, and the key must be shared between the nodes in a secure, atomic way to prevent the network to enter an irrecoverable state.

Inclusing the entry port in the checksum would also be an appreciable increase in security, since it increses the number of targets an attacker would need to breach at once to be able to break routing.

Bibliography

[1] P. Bosshart *et al.*, “P4: programming protocol-independent packet processors,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014, doi: 10.1145/2656877.2656890.

[2] C. Dominicini *et al.*, “PolKA: Polynomial Key-based Architecture for Source Routing in Network Fabrics,” in *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, 2020, pp. 326–334. doi: 10.1109/NetSoft48620.2020.9165501.

[3] E. S. Borges *et al.*, “PoT-PolKA: Let the Edge Control the Proof-of-Transit in Path-Aware Networks,” *IEEE Transactions on Network and Service Management*, vol. 21, no. 4, pp. 3681–3691, 2024, doi: 10.1109/TNSM.2024.3389457.

[4] C. Dominicini *et al.*, “Deploying PolKA Source Routing in P4 Switches : (Invited Paper),” in *2021 International Conference on Optical Network Design and Modeling (ONDM)*, 2021, pp. 1–3. doi: 10.23919/ONDM51796.2021.9492363.

[5] R. Fontes, S. Afzal, S. Brito, M. Santos, and C. Esteve Rothenberg, “Mininet-WiFi: Emulating Software-Defined Wireless Networks,” in *2nd International Workshop on Management of SDN and NFV Systems, 2015(ManSDN/NFV 2015)*, Nov. 2015.

[6] 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, in Hotnets-IX. Monterey, California: Association for Computing Machinery, 2010. doi: 10.1145/1868447.1868466.

[7] "Behavioral Model." Accessed: Sep. 26, 2024. [Online]. Available: https://github.com/p4lang/behavioral-model

[8] Wireshark Foundation, "Wireshark." Accessed: Sep. 26, 2024. [Online]. Available: https://www.wireshark.org/

[9] B. P. secdev, "Scapy." Accessed: Sep. 26, 2024. [Online]. Available: https://scapy.net/

[10] J.-P. Aumasson and D. J. Bernstein, "SipHash: a fast short-input PRF." [Online]. Available: https://eprint.iacr.org/2012/351

[11] M. Stigge, H. Plötz, W. Müller, and J.-P. Redlich, "Reversing crc–theory and practice." 2006.

[12] M. Martinello *et al.*, "PathSec: Path-Aware Secure Routing with Native Path Verification and Auditability." 2024.