



**Universidade Federal do Espírito Santo**  
**Colegiado do Curso de Ciência da Computação**  
**Trabalho de Conclusão de Curso I**  
**Opção 2: Desenvolvimento de um Protótipo**  
**RELATÓRIO DE RESULTADOS PARCIAIS**

**Estudante: Henrique Coutinho Layber**

**Orientador: Roberta Lima Gomes**

## **1. INTRODUÇÃO**

Este trabalho tem como objetivo apresentar uma implementação de um mecanismo de sondagem e validação de rota para o protocolo de roteamento de origem (Source Routing) PolKA, com ênfase em redes definidas por software (Software Defined Networks). O mecanismo é baseado em uma composição de funções de checksum em switches de núcleo, que permite a uma parte confiável verificar se um pacote percorreu a rede ao longo do caminho definido pela origem. A validação garante que a rota está funcionando e está configurada corretamente.

O projeto foi desenvolvido como parte do Trabalho de Conclusão de Curso (TCC), que visa integrar os conhecimentos adquiridos ao longo da graduação em uma aplicação prática e relevante. O sistema é capaz de corretamente invalidar pacotes que sofrem de adição de switches, detours e outras formas de ataques, e é capaz de validar corretamente pacotes que seguem o caminho correto. Será elaborada uma extensão para uma conferência na área de redes, e é também uma contribuição para um sistema com desenvolvimento em andamento, PathSec[1].

# An Implementation of Route Probing and Validation on PolKA

Henrique Coutinho Layber<sup>†</sup>, Roberta Lima Gomes<sup>†</sup>, Magno Martinello<sup>†</sup>, Vitor B. Bonella<sup>†</sup>,  
Everson S. Borges<sup>‡</sup>, Rafael Guimarães<sup>†‡</sup>

<sup>†</sup>Department of Informatics, Federal University of Espírito Santo

<sup>‡</sup>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 source routing protocol PolKA. 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[2] implementation for a new protocol layer for PolKA[3], able to do validate the actual route used for a packet, and is available on GitHub<sup>1</sup>. 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_ids` independently.

## 2. Problem Definition

Let  $i$  be the source node (ingress node) and  $e$  be the destination node (egress node). Let path  $P_{i \rightarrow e}$  be a sequence of nodes:

$$P_{i \rightarrow e} = (i, s_1, s_2, \dots, s_{n-1}, s_n, e)$$

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  $i$  [4].  $i$  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

---

<sup>1</sup><https://github.com/Henriquelay/PolKA-halfsiphash/tree/remake/mininet/PolKA-example>

Number System (RNS)[5], 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  $P_{i \rightarrow e}$ , and a captured sequence of switches actually traversed  $P_j$ , a solution should identify if  $P_{i \rightarrow e} = P_j$ .

### 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  $f(k_{s_i}, x) = f_{s_i}(x)$ . 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:

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

$s_i$   $i$ -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. Every link is assumed to be perfect, that is, no packet loss, no packet corruption, and no packet duplication.
3. Protocol boundary is IPv4. This means that PolKA is only used inside this network, and only IPv4 is used outside.

#### 3.2. Setup

All implementation and experiments took place on a VM<sup>2</sup> setup with Mininet-wifi[6], and were targeting Mininet's[7] Behavioral Model version 2 (BMv2)[8]. Wireshark[9] was used to analyze packets, and Scapy[10] was used to parse packets programatically and automatic tests.

---

<sup>2</sup>Available on PolKA's repository <https://github.com/nerds-ufes/PolKA>

### 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 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.

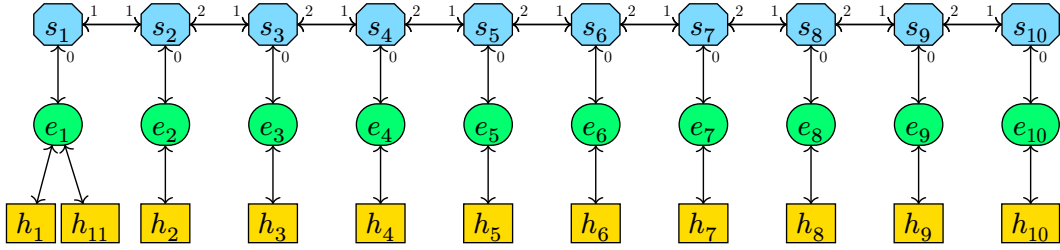


Figure 1: Topology setup.

$s_n$  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.

The implementation for parsing on edge nodes can be found in Appendix A.1, and for core nodes in Appendix A.2.

#### 3.3.2. Encapsulation

PolKA headers consists of the route polynomial (`routeid`), along with `version`, `tll` 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.

The implementation for encapsulation and related headers can be found in Appendix B.

### 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:

$$l\_hash \leftarrow CRC_{32}(\text{exit\_port} \oplus l\_hash \oplus \text{node\_id})$$

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<sup>3</sup>.

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

The implementation for a node doing an individual checksum calculation step can be found in Appendix C.

### 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.

The implementation for decapsulation can be found in Appendix D.

## 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. In the examples, we are measuring  $P_{e_1 \rightarrow e_{10}} = (e_1, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, e_{10})$

---

<sup>3</sup><https://reveng.sourceforge.io/crc-catalogue/all.htm#crc.cat.crc-32-iso-hdlc>

<sup>4</sup>[https://github.com/Henriquelay/PolKA\\_probe\\_checker/](https://github.com/Henriquelay/PolKA_probe_checker/)

<sup>5</sup><https://github.com/mrhoray/crc-rs>

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

Table 1: Packet trace while traversing  $P_{e_1 \rightarrow e_{10}}$ .

### 3.5. Adversity scenarios

The solution has been tested against some adversarial scenarios, and checked against the same initial seed but under the base topology as described on Figure 1.

#### 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.

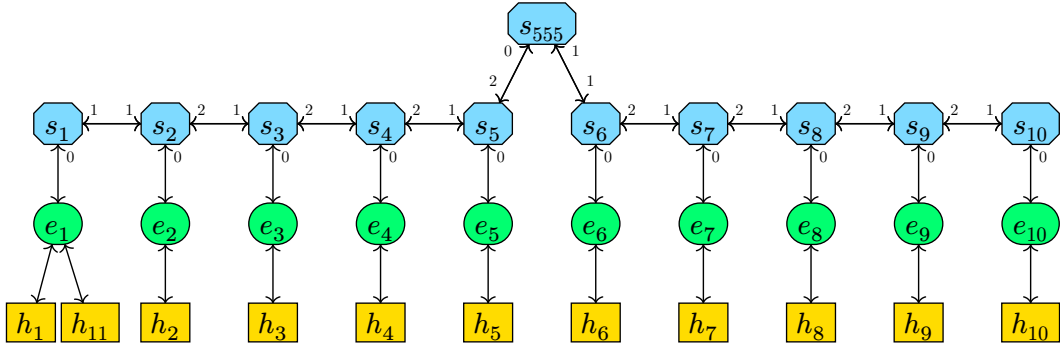


Figure 2: Topology setup for addition scenario.

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

Table 2: Packet trace while traversing  $P_{e_1 \rightarrow e_{10}}$  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.

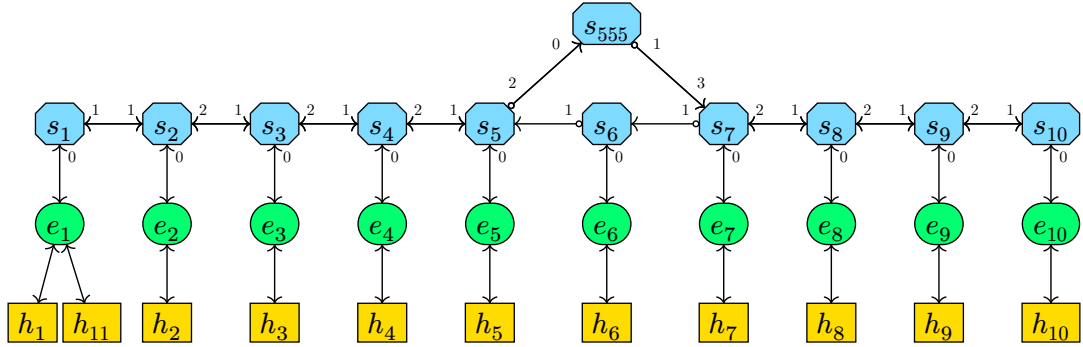


Figure 3: Topology setup for detour scenario.

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

Table 3: Packet trace while traversing  $P_{e_1 \rightarrow e_{10}}$  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.

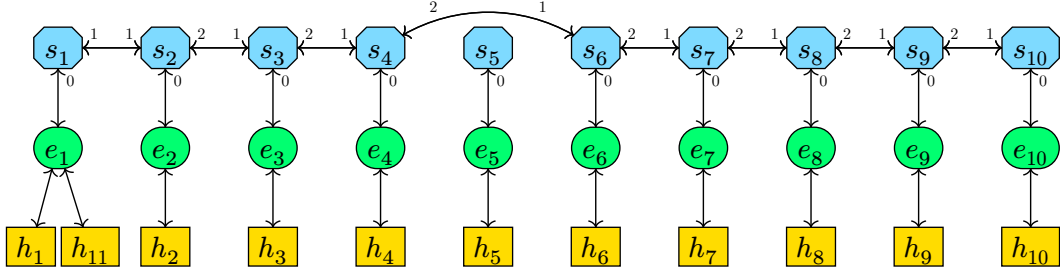


Figure 4: Topology setup for skipping scenario.

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

Table 4: Packet trace while traversing  $P_{e_1 \rightarrow e_{10}}$  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[11], to be used instead of the  $\text{CRC}_{32}$ . This would make the system more secure, since  $\text{CRC}_{32}$  is a well-known as a checksum function that can be easily reversed[12]. 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 be integrated into PathSec[1], 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.

Including 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 the same time to be able to alter the path.

A timing analysis and stress tests can both be done to check if the incurred overhead is acceptable for the network. This is important, since the network must be able to handle the increased load without dropping packets. A more robust solution would be to use a more secure hash function, such as SHA-256, but this would increase the overhead of the network, and would require a more powerful hardware to be able to handle the increased load.

## 5. Conclusion

As the examples show, our solution is able to detect when a packet is not following the expected path for most cases, and can be used to detect misconfigurations in the network. The solution is not perfect, but it is a step in the right direction for a more secure network.

## Bibliography

- [1] M. Martinello *et al.*, “PathSec: Path-Aware Secure Routing with Native Path Verification and Auditability.” 2024.
- [2] 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](https://doi.org/10.1145/2656877.2656890).

- [3] 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](https://doi.org/10.1109/NetSoft48620.2020.9165501).
- [4] 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](https://doi.org/10.1109/TNSM.2024.3389457).
- [5] 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](https://doi.org/10.23919/ONDM51796.2021.9492363).
- [6] 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.
- [7] 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](https://doi.org/10.1145/1868447.1868466).
- [8] “Behavioral Model.” Accessed: Sep. 26, 2024. [Online]. Available: <https://github.com/p4lang/behavioral-model>
- [9] Wireshark Foundation, “Wireshark.” Accessed: Sep. 26, 2024. [Online]. Available: <https://www.wireshark.org/>
- [10] B. P. secdev, “Scapy.” Accessed: Sep. 26, 2024. [Online]. Available: <https://scapy.net/>
- [11] J.-P. Aumasson and D. J. Bernstein, “SipHash: a fast short-input PRF.” [Online]. Available: <https://eprint.iacr.org/2012/351>
- [12] M. Stigge, H. Plötz, W. Müller, and J.-P. Redlich, “Reversing crc—theory and practice.” 2006.

## Appendix

### A. Parsers

#### A.1. Edge Node

```

1  parser MyParser(
2      packet_in packet,
3      out headers hdr,
4      inout metadata meta,
5      inout standard_metadata_t standard_metadata
6  ) {
7      state start {
8          transition parse_ethernet;
9      }
10
11     state parse_ethernet {
12         // Reads and pops the header. We will need to `.emit()` it back later

```

p4

```

13     packet.extract(hdr.ethernet);
14     transition select(hdr.ethernet.ethertype) {
15         // If the packet comes from outside (ethernet packet)
16         TYPE_IPV4: parse_ipv4;
17
18         // If the packet comes inside (PolKA packet)
19         TYPE_POLKA: parse_PolKA;
20
21         // Any other packet
22         default: accept;
23     }
24 }
25
26 state parse_PolKA {
27     packet.extract(hdr.PolKA);
28
29     transition select(hdr.PolKA.version) {
30         PROBE_VERSION: parse_PolKA_probe;
31         // Any other packet
32         default: parse_ipv4;
33     }
34 }
35
36 state parse_PolKA_probe {
37     packet.extract(hdr.PolKA_probe);
38     transition parse_ipv4;
39 }
40
41 state parse_ipv4 {
42     packet.extract(hdr.ipv4);
43     transition accept;
44 }
45 }

```

## A.2. Core Node

```

1  parser MyParser(
2      packet_in packet,
3      out headers hdr,
4      inout metadata meta,
5      inout standard_metadata_t standard_metadata
6  ) {
7      state start {
8          meta.apply_sr = 0;
9          transition verify_ethernet;
10     }
11

```

```

12     state verify_ethernet {
13         packet.extract(hdr.ethernet);
14         transition select(hdr.ethernet.ethertype) {
15             TYPE_POLKA: get_PolKA_header;
16             // Should be dropped when apply_sr is 0
17             // But can't use drop here on BMV2
18             default: accept;
19         }
20     }
21
22     state get_PolKA_header {
23         meta.apply_sr = 1;
24         packet.extract(hdr.PolKA);
25         meta.route_id = hdr.PolKA.routeid;
26
27         transition select(hdr.PolKA.version) {
28             PROBE_VERSION: parse_PolKA_probe;
29             // Any other packet
30             default: accept;
31         }
32     }
33
34     state parse_PolKA_probe {
35         packet.extract(hdr.PolKA_probe);
36         transition accept;
37     }
38 }

```

## B. Encapsulation

```

1  control TunnelEncap(
2      inout headers hdr,
3      inout metadata meta,
4      inout standard_metadata_t standard_metadata
5  ) {
6      action tdrop() {
7          mark_to_drop(standard_metadata);
8      }
9
10     action add_sourcerouting_header (
11         bit<9> port,
12         bit<1> sr,
13         mac_addr_t dmac,
14         PolKA_route_t routeIdPacket
15     ){
16         // Has to be set to valid for changes to be committed

```

```

17     hdr.PolKA.setValid();
18
19     hdr.PolKA.version = REGULAR_VERSION;
20     hdr.PolKA.ttl = 0xFF;
21
22     meta.apply_sr = sr;
23     standard_metadata.egress_spec = port;
24     hdr.PolKA.routeid = routeIdPacket;
25     hdr.ethernet.dst_mac_addr = dmac;
26
27     hdr.PolKA.proto = TYPE_POLKA;
28     // Replicating on both headers for consistency
29     hdr.ethernet.ethertype = TYPE_POLKA;
30 }
31
32 // Adds a PolKA header to the packet
33 // Table name can't be changed because it is the name defined by node
configuration files
34 table tunnel_encap_process_sr {
35     key = {
36         hdr.ipv4.dst_addr: lpm;
37     }
38     actions = {
39         // Actions names also can't be changed because they are the names
defined by node configuration files
40         add_sourcerouting_header;
41         tdrop;
42     }
43     size = 1024;
44     default_action = tdrop();
45 }
46
47 apply {
48     tunnel_encap_process_sr.apply();
49
50     if (meta.apply_sr == 0) {
51         hdr.PolKA.setInvalid();
52     } else {
53         // Not needed - it is already set to valid in inside match arm
54         //     hdr.PolKA.setValid();
55     }
56 }
57 }

```

## C. Checksum Calculation

```
1  const bit<16> TYPE_IPV4 = 0x0800;
```

```

2  const bit<16> TYPE_POLKA = 0x1234;
3
4  const bit<8> REGULAR_VERSION = 0x01;
5  const bit<8> PROBE_VERSION = 0xF1;
6
7  #include hdr_ethernet.p4"
8  #include "hdr_ipv4.p4"
9  #include "hdr_PolKA.p4"
10
11 struct metadata {
12     bit<1>    apply_sr;
13     bit<9>    port;
14     bit<16>   switch_id;
15     PolKA_route_t route_id;
16 }
17
18 header PolKA_probe_t {
19     bit<32> timestamp;
20     bit<32> l_hash;
21 }
22
23 struct headers {
24     ethernet_t    ethernet;
25     PolKA_t       PolKA;
26     PolKA_probe_t PolKA_probe;
27     ipv4_t        ipv4;

```

```

1  control MySwitchId(
2      inout headers hdr,
3      inout metadata meta
4  ) {
5      action switchid (
6          bit<16> switch_id
7      ){
8          meta.switch_id = switch_id;
9      }
10
11     // Adds a PolKA header to the packet
12     // Table name can't be changed because it is the name defined by node
13     table config {
14         key = {
15             meta.apply_sr: exact;
16         }
17         actions = {

```

```

18         // Actions names also can't be changed because they are the names
   defined by node configuration files
19         switchid;
20     }
21     size = 128;
22 }
23
24     apply {
25         meta.apply_sr = 0;
26         config.apply();
27         hdr.PolKA.ttl = meta.switch_id[7:0];
28     }
29 }
30
31 control MySignPacket(
32     inout headers hdr,
33     inout metadata meta
34 ) {
35     // Signs the packet
36     apply {
37         // Gets the routeId and installs it on meta.route_id
38         MySwitchId.apply(hdr, meta);
39         hdr.PolKA_probe.setValid();
40
41         // At this point, `meta.port` should be written on already
42         hdr.PolKA_probe.l_hash = (bit<32>) meta.port ^ hdr.PolKA_probe.l_hash ^
   (bit<32>) meta.switch_id;
43
44         bit<16> nbase = 0;
45         bit<32> min_bound = 0;
46         bit<32> max_bound = 0xFFFFFFFF;
47         hash(
48             hdr.PolKA_probe.l_hash,
49             HashAlgorithm.crc32,
50             min_bound,
51             {hdr.PolKA_probe.l_hash},
52             max_bound
53         );
54     }
55 }

```

## D. Decapsulation

```

1 control MyIngress(
2     inout headers hdr,
3     inout metadata meta,
4     inout standard_metadata_t standard_metadata

```

```
5  ) {
6      // Removes extra headers from PolKA packet, leaves it as if nothing had
    touched it.
7      action tunnel_decap() {
8          // Set ethertype to IPv4 since it is leaving PolKA
9          hdr.PolKA.proto = TYPE_IPV4;
10         // Replicating on second header for consistency
11         hdr.ethernet.ethertype = TYPE_IPV4;
12
13         // Does not serialize routeid
14         hdr.PolKA.setInvalid();
15
16         // Should be enough to "decap" packet
17
18         // In this example, port `1` is always the exit node
19         standard_metadata.egress_spec = 1;
20     }
21
22     apply {
23         if (hdr.ethernet.ethertype == TYPE_POLKA) {
24             // Packet came from inside network, we need to make it a normal pkt
25             tunnel_decap();
26         } else {
27             // Packet came from outside network, we need to make it a PolKA pkt
28             TunnelEncap.apply(hdr, meta, standard_metadata);
29         }
30         MyProbe.apply(hdr, meta);
31     }
32 }
```