eBPF-based Ordered Proof of Transit for Trustworthy Service Function Chaining

Takanori Hara, *Member, IEEE,* and Masahiro Sasabe, *Member, IEEE*

***Abstract*—Service function chaining (SFC) establishes a service path where a sequence of functions is executed according to service requirements. However, SFC lacks a mechanism to ensure proper traversal of relay nodes in the data plane. Misconfigurations and the presence of attackers can lead to forwarding anomalies and path deviation, potentially allowing packets to bypass security network functions in the service path. To mitigate potential security breaches, ordered proof of transit (OPoT) has been proposed as a mechanism to verify whether traffic adheres to the designated path. In this paper, we realize lightweight OPoT-based path verification based on extended Berkeley Packet Filter (eBPF) for trustworthy SFC. Furthermore, by integrating it with the existing SFC proxy, we extend the proposed approach to accommodate both SFC-aware and SFC-unaware virtual network functions (VNFs) in the segment routing over IPv6 data plane (SRv6) domain. Through experiments, we demonstrate the capability of the proposed approach to detect path deviations. Additionally, we reveal the performance limitations of the proposed approach.**

***Index Terms*—Service Function Chaining (SFC), extended Berkeley Packet Filter (eBPF), Ordered Proof-of-Transit (OPoT), Segment Routing over IPv6 Data Plane (SRv6), SFC proxy**

I. Introduction

Thanks to the integration of *network functions virtualization* (NFV) and *software defined networking* (SDN), a network operator can deploy a *network slice*, a logically isolated network, to meet diverse requirements of applications and services. *Service function chaining* (SFC) is a fundamental technology to establish a service path that traverses executable virtual network functions (VNFs) according to the service chain requirements [1], [2]. *OpenFlow* [3], *segment routing over IPv6 data plane* (SRv6) [4], [5] and its extension with *compressed segment identifiers (cSIDs)* [6], [7], *segment routing over multi-protocol label switching* (SR-MPLS) [5], [8], and *network service header* (NSH) [9] are utilized for implementing SFC.

Despite its flexibility in accommodating diverse service requirements, SFC encounters challenging security issues [10]–[13] due to absence of a mechanism to verify traffic adherence to the specified service path while traversing VNFs in the prescribed sequence. These security concerns, such as network operators' misconfigurations and potential attacks, may lead to forwarding anomalies and path deviations, enabling traffic to bypass security network functions in the service path. Consequently, this not only breaches end-user contracts and policies but also poses potential security risks by bypassing security VNFs, such as firewalls and deep packet inspectors, integrated into the service chain.

Therefore, both end-users and network operators require a mechanism to verify that traffic has traversed all VNFs in the service chain in the intended order. There have been several mechanisms to detect forwarding anomalies and path deviations [14]–[16]. *Proof of transit* (PoT) [14] is a mechanism to verify whether the traffic traverses a valid set of relay nodes by applying *Shamir's secret sharing* (SSS) [17]. *Ordered PoT* (OPoT) [14] integrates symmetric masking with PoT, enabling the verification of packet traversal of relay nodes in appropriate order. *In-situ Operations, Administration, and Maintenance* (IOAM) [15] is a general framework to collect network telemetry data within the packet while the packet traverses a particular network domain. IOAM's PoT option type can support PoT-based SFC verification. *ChainSign* provides higher security than OPoT but requires SFC-aware VNFs that support SFC-related packets (e.g., SRv6 packets) [16].

In this paper, we propose OPoT verification using *extended Berkeley Packet Filter* (eBPF) [18], which allows programs strictly verified by an eBPF verifier to run in the Linux kernel space. To the best of our knowledge, this paper presents the first implementation of OPoT using eBPF. Implementing path verification functions in user-level VNFs complicates VNF functionality, which hinders low-cost software updates. The proposed mechanism facilitates software modularization by decoupling the path verification functions from user-level VNFs and offloading them to eBPF as kernel network functions. Additionally, some VNFs may be SFC-unaware (e.g., legacy VNFs). Supporting these SFC-unaware VNFs provides network operators with several benefits: (1) leveraging previous investments during migration, (2) facilitating interoperability and multi-vendor environments, and (3) reducing the costs of implementing SFC-aware VNFs [19]. We develop a mechanism applicable to SRv6 environments containing both SFC-aware and SFC-unaware VNFs by integrating OPoT with an existing eBPF-based SFC proxy [20]. Through experiments, we demonstrate the proposed scheme's ability to accurately verify OPoT-based service paths and assess the throughput impact associated with its introduction.

The main contributions of the manuscript are as follows:

1) By offloading OPoT verification to eBPF, the proposed mechanism ensures path verification without modifying user-level VNFs. This approach enhances software modularity and promotes code reusability. Furthermore, the proposed OPoT verification can accomodate both SFC-unaware VNFs and kernel network functions by

This work was supported in part by the Japan Society for the Promotion of Science (JSPS) KAKENHI (B) under Grant 23K24842 and the JSPS Grant-in-Aid for Young Scientists under Grant 23K16869.

T. Hara is with the Division of Information Science, Nara Institute of Science and Technology, Nara, Japan, e-mail: hara@ieee.org. M. Sasabe is with the Faculty of Informatics, Kansai University, Takatsuki, Japan, email: m-sasabe@ieee.org.

implementing OPoT using eBPF and integrating it with the existing eBPF-based SFC proxy [20]. To the best of our knowledge, this is the first work to implement eBPF-based OPoT.

2) Through experiments, we demonstrate that both the eBPF-based OPoT and its integration with eBPF-based SFC proxy function properly.
3) The experiment results reveal the performance limitations of the proposed OPoT verification in terms of packet rate, bit rate, and processing overhead. The eBPF-based OPoT schemes nearly achieve the line-rate performance up to a maximum link speed of up to 3 Gbps. At a maximum link speed of 5 Gbps, the performance degradation of the eBPF-based OPoT and the eBPF-based OPoT with cSIDs is evaluated to 7.0% and 8.3%, respectively, compared with SRv6 and SRv6 with cSIDs transmission, primarily due to limitations in packet processing capability. These results imply that the path verification using the eBPF-based OPoT is practical for many use cases in terms of its performance.

The rest of the manuscript is organized as follows. Section II gives an overview of related work. In Section III, we introduce PoT, SRv6, eBPF, and SFC proxy. Section IV presents the proposed path verification method for trustworthy SFC using eBPF, SRv6, SFC proxy, and OPoT. Section V demonstrates the fundamental characteristics of the proposed scheme. Finally, Section VI gives the conclusion.

II. Related Work

There have been studies on establishing trustworthy SFC [13]–[16], [21], [22]. PoT is a mechanism to verify whether traffic follows a designated path by applying SSS [14], which can be implemented in the IOAM [15]. PoT can be applied to traffic engineering and policy-based routing in addition to SFC. In PoT, all packets on the service path are associated with PoT metadata, which is updated by a piece of secret whenever each packet passes through a node and subsequently used for reconstruction of secret at the verifier node. Furthermore, OPoT integrates symmetric masking with PoT to realize verification of the routing order of each node on the service path [14]. This mechanism is compatible with source routing (e.g., SRv6 and SR-MPLS) and facilitates the aggregation of routing information.

PoT has been implemented in several network-programmable frameworks [23], [24]. In [23], Borges et al. proposed a PoT scheme for source routing using a polynomial key-based architecture [25], implemented on dedicated hardware (e.g., programmable switches using the *Programming Protocol-Independent Packet Processors* (P4) language. The Fast Data Project also incorporated PoT in a *Vector Packet Processing* (VPP) framework, enabling kernel-bypass packet processing on commodity hardware [24]. Kernel-bypass packet processing requires constructing and managing a network stack in the user space, thus avoiding the kernel network stack's overhead. In contrast, our study implements an in-kernel PoT scheme via an eBPF program operating on commodity hardware. This in-kernel packet processing approach allows the scheme to be compatible with VNFs running on both user and kernel spaces. A survey on path verification mechanisms is provided in [26].

The PoT draft addresses security considerations against representative risks, including partial and full detours, eavesdropping on PoT metadata, and replay attack [14]. PoT can detect partial and full detours as long as the following conditions are met: the secret information is not leaked, and polynomials are of $k-1$th degree when k nodes are present in the path. To attempt to predict the secret information, passive attackers may try to eavesdrop on PoT metadata for differential analysis by adding a node in the original path. Section 7.2 of [14] discusses that predicting the secret information through differential analysis is difficult. In a reply attack, a passive attacker can reuses PoT metadata from an old packet to replay it with a new packet, bypassing PoT reconstruction. It has been shown that such attacks can be prevented by appending a timestamp (i.e., a sequence number) to the random value in the PoT metadata. The verifier node can employ existing anti-replay mechanisms (e.g., sliding window in IPsec) to detect replayed PoT metadata. For further details on these countermeasures, see Section 7 of [14].

Several studies have also been conducted to further enhance security. Aguado et al. improved the security of key exchange techniques and random value generation for PoT by applying quantum key distribution and quantum random number generators to PoT [21]. Pattaranantakul et al. proposed a protocol, named ChainSign, using ordered multi-signature (OMS) and NSH [9] to ensure higher security than the PoT [16]. ChainSign improves both security and computational complexity of PoT but requires SFC-aware VNFs supporting both OMS and NSH. This implicitly assumes that all VNFs on the network must be SFC-aware.

The challenging issue of the computational complexity in OPoT is that computational processing is required per packet whenever a packet passes through a node on the service path. In this paper, we aim to establish lightweight and fast packet processing by implementing OPoT in the eBPF program running on the Linux kernel. eBPF is a technology to control the Linux kernel by injecting strictly verified programs into it [18], [27]. Thanks to its lightweight, extensibility, and portability, eBPF is applied to various networking domains: packet filtering [28], SFC proxy [20], SRv6 [29], and VNFs [30], [31]. The SFC proxy is a technology that enables the incorporation of SFC-unaware VNFs into SFC [20]. The proposed scheme realizes trustworthy SFC by combining OPoT with the SFC proxy even in environments with a mixture of SFC-aware and unaware VNFs. Furthermore, since the proposed scheme is implemented in the eBPF program, it is applicable not only to VNFs operating in user space but also to those operating in kernel space [30]. Unlike the existing implementation of the eBPF-based SFC proxy for SR-MPLS [20], this paper implements the SFC proxy for SRv6, one of the most popular source routing schemes.

III. Foundational Technologies

Existing trustworthy SFC schemes do not support kernel network functions and SFC-unaware VNFs [14], [16], [23],

TABLE I: Notations.

Symbol	Description
$\mathcal{L}$	Set of SR policies
$\mathcal{V}$	Set of nodes
π_l	Service path of SR policy $l \in \mathcal{L}$, $\pi = (v_0, v_1, \dots, v_{k-1})$
f	Polynomial
$\boldsymbol{a}$, $\boldsymbol{b}$	Polynomial coefficient vectors
p	Prime number
s	Secret
$s_1, \dots, s_n$	n pieces of secret, i.e., shares
r	Random value
LPC_j	Lagrange polynomial constant of the jth node
CML_j	Cumulative value of the jth node
(x_j, y_j)	Point of the jth node
$\boldsymbol{c}_j$	Cipher keys, $\boldsymbol{c}_j = (c_j^r, c_j^{\mathrm{CML}})$
$\boldsymbol{d}_j$	Decipher keys, $\boldsymbol{d}_j = (d_j^r, d_j^{\mathrm{CML}})$
$\mathrm{P}_{l,j}$	OPoT parameter

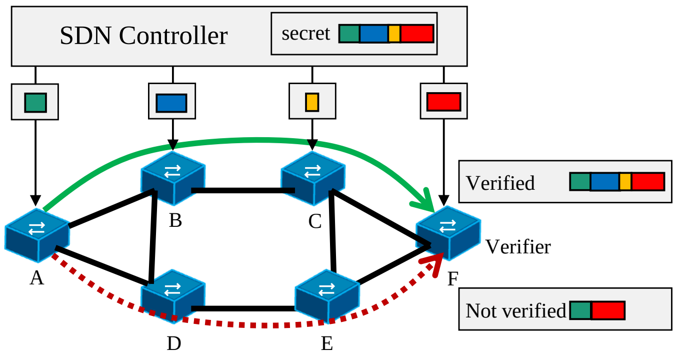


Fig. 1: Overview of PoT.

[24]. This section outlines the fundamental technologies of the proposed scheme, focusing on OPoT, SRv6, eBPF, and SFC proxy, to address these limitations. To achieve trustworthy SFC, OPoT verification is implemented to verify whether traffic follows a designated service path. (Details are provided in Section III-A.) SFC enforces traffic to traverse a specific service path using SRv6, as discussed in Sections III-B. Since the existing OPoT verification is incompatible with kernel network functions, we implement OPoT verification in eBPF, mentioned in Section III-C. To apply OPoT verification to both SFC-aware and SFC-unaware VNFs, the proposed scheme integrates the eBPF-based OPoT verification with the existing eBPF-based SFC proxy, detailed in Section III-D. Table I presents the notations used in this paper.

A. Proof of Transit

Fig. 1 illustrates the overview of PoT [14]. It aims to verify whether traffic traverses "Path 1" (i.e., $A \rightarrow B \rightarrow C \rightarrow F$) as depicted in Fig. 1. In PoT, the SDN controller possesses the *secret* s of "Path 1" and distributes a piece of the secret s (i.e., *share*) to each node on "Path 1." The traffic retrieves the share from a node on "Path 1" after passing through it. If all shares are acquired upon the traffic arrival at the destination node (verifier node), it can be demonstrated that the traffic has traversed all nodes on "Path 1". Conversely, if the traffic follows "Path 2" (i.e., $A \rightarrow D \rightarrow E \rightarrow F$) instead of "Path 1," the verifier node can confirm that the traffic fails to obtain all shares. In PoT, this share retrieval process is verified through mathematical means. Sections III-A1 and III-A2 elaborate on SSS and security enhancement of PoT, respectively. Section III-A3 provides the reconstruction of the secret s, while Section III-A4 introduces OPoT, which incorporates the ordering property into PoT.

1) Principle of Shamir's Secret Sharing: SSS is a cryptographic algorithm that distributes a secret among multiple nodes [17]. Under arithmetic modulo a prime number p, SSS partitions the secret s into n pieces (shares) $s_1, s_2, \dots, s_n$, and can reconstruct the secret s if and only if k $(1 \le k \le n)$ or more shares are collected. PoT adopts Lagrange interpolation to realize SSS by assuming $k = n$. Given distinct k points (x_j, y_j) $(j = 0, 1, \dots, k-1)$, the Lagrange interpolation approximates a curve connecting them to a polynomial $f(x) = \sum_{j=0}^{k-1} a_j x^j$ of the $k-1$th degree:

$$f(x) = \sum_{j=0}^{k-1} y_j l_j(x), \tag{1}$$

where $l_j(x)$ denotes a Lagrange basis polynomial, which is given by

$$l_j(x) = \prod_{0 \le m \le k-1, m \ne j} \frac{x - x_m}{x_j - x_m}. \tag{2}$$

Considering the fact the constant term $f(0)$ in Eq. (1) can be divided into k constant terms $y_j l_j(0)$ in Eq. (2), $f(0)$ can be reconstructed as a linear sum under arithmetic modulo a prime number p:

$$f(0) \bmod p = \left(\sum_{j=0}^{k-1} (y_j \mathrm{LPC}_j \bmod p) \right) \bmod p, \tag{3}$$

where $\mathrm{LPC}_j = l_j(0) \bmod p$ $(j = 0, 1, \dots, k-1)$. In the PoT, $f(0)$ is regarded as a secret s and k shares (i.e., (x_j, y_j) and LPC_j) $(j = 0, \dots, k-1)$ are distributed among k nodes $(v_0, \dots, v_{k-1})$ that execute the corresponding VNF on the service path $\pi = (v_0, \dots, v_{k-1})$.

2) Security Improvement of PoT with Randomness: Theoretically, PoT can be realized using only the mechanism mentioned in Section III-A1. However, using the same polynomial for each packet poses a potential security risk. To tackle these concerns, PoT incorporates a polynomial $y = f_1(x) = s + \sum_{j=1}^{k-1} a_j x^j$ of $k-1$th degree defined per service path and a polynomial $z = f_2(x, r) = r + \sum_{j=1}^{k-1} b_j x^j$ of the $k-1$th degree defined per packet, where r is generated by the ingress node v_0 of SFC upon the arrival of a new packet. The polynomial f_1 is kept confidential, while only the 64-bit random value r of the polynomial f_2 is observable by all nodes, including potential attackers. The sum of two polynomials (i.e., $f_3(x, r) = f_1(x) + f_2(x, r)$) results in a polynomial defined per packet, containing the secret s. The SDN controller distributes the points (x_j, y_j) on the polynomial f_1, LPC_j, coefficient vector $\boldsymbol{b} = (b_1, \dots, b_{k-1})$, and the prime number p to each node $v_j \in \pi$ on the service path.

3) Reconstruction: Considering that Eq. (3) can be computed sequentially, the PoT performs sequential computation, stores the result in the packet header as 64-bit CML, and forwards the packet to the next node upon traversal. At each node $v_j \in \pi$, the value of CML (i.e., CML_j) is updated using the following equation and stored in the header along with a random value r.

$$\mathrm{CML}_j = \mathrm{CML}_{j-1} + (y_j + z_j)\mathrm{LPC}_j \bmod p, \quad (4)$$

where $\mathrm{CML}_{-1} = 0$ at the origin node v_0 and r is generated as a random value. Other nodes v_j retrieve CML_{j-1} and r from the packet header. This ensures that the header size remains constant, regardless of the value of k. After calculating CML_{k-1} using Eq. (4), the last node v_{k-1} further updates it with $\mathrm{CML}_{k-1} \bmod p$.

If the packet correctly follows the designated service path, the kth node (i.e., verifier node) obtains the constant term $\mathrm{CML}_{k-1} = a_0 + b_0 \bmod p$ of polynomial f_3 from Eq. (3). If $\mathrm{CML}_{k-1} = (r+s) \bmod p$, the verifier node can confirm that the packet has traversed the designated k nodes. However, POT cannot verify the order of visited nodes.

4) Ordered PoT: OPoT can verify the order of visited nodes by applying symmetric masking (i.e., XOR cipher) to the PoT metadata [14]. The SDN controller distributes two 64-bit cipher keys $(c^r_j, c^{\mathrm{CML}}_j)$ for ciphering PoT metadata (r, CML_j) and two 64-bit decipher keys $(d^r_j, d^{\mathrm{CML}}_j)$ for deciphering PoT metadata (r, CML_{j-1}) to the nodes $\forall v_j \in \pi$ along the service path. The decipher (cipher) keys of node v_j are set to the cipher (resp. decipher) keys of preceding node v_{j-1} (resp. succeeding node v_{j+1}). Note that the origin (resp. destination) node v_0 (resp. v_{k-1}) does not have the cipher (resp. decipher) keys. In OPoT, node v_j deciphers the PoT metadata using the XOR operation upon packet arrival, then ciphers the PoT metadata using the XOR operation and forwards the packet to the next node.

Fig. 2 illustrates an example of the OPoT reconstruction process. Suppose a valid path is given as (v_1, v_2, v_3, v_4). The upper part of Fig. 2 shows a successful OPoT verification case, while the middle part shows a failure due to an invalid order of the same nodes, (v_1, v_2, v_3, v_4). To highlight the difference between OPoT and PoT, the lower part of Fig. 2 shows that PoT verification succeeds even for the invalid path (v_1, v_2, v_3, v_4). We assume $s = 19$, $p = 53$, $r = 10$, $f_1(x) = 9x^3 + 10x^2 + 6x + s$, and $f_2(x, r) = 10x^3 + 8x^2 + 3x + r$. Each node v_j $(j = 1, \ldots, 4)$ has parameters $(x_j, y_j, z_j, \mathrm{LPC}_j)$, the cipher key $(c^r_j, c^{\mathrm{CML}}_j)$, and the decipher key $(d^r_j, d^{\mathrm{CML}}_j)$. Recall that z_j is computed using a deciphered random value r in the received packet. In the upper part of Fig. 2, if node v_2 receives a packet from preceding node v_1, the PoT metadata is successfully deciphered using the symmetric key $(18, 44)$ shared between v_1 and v_2. As a result, the deciphered value $\mathrm{CML}_1 \oplus c^{\mathrm{CML}}_1 \oplus d^{\mathrm{CML}}_2$ (resp. $r \oplus c^r_1 \oplus d^r_2$) computed by v_2 matches the value of CML_1 (resp. r) calculated by v_1. In the middle part of Fig. 2, if node v_3 receives a packet from v_1, it fails in decryption because the cipher and decipher keys differ (i.e., $(18, 44)$ and $(36, 123)$). The failed decryption at node v_j yields invalid values of CML_j and z_j. Specifically, r is 10, but the deciphered value $r \oplus c^r_1 \oplus d^r_3$ computed by v_3 is 40, leading to an incorrect calculation of $z_3 = f_2(x_3, r)$. Similarly, the CML_1 value calculated at node v_1 is 20, but the deciphered value $\mathrm{CML}_1 \oplus c^{\mathrm{CML}}_1 \oplus d^{\mathrm{CML}}_3$ computed by node v_3 is 34. These invalid random and CML values propagate incorrect information to succeeding nodes, ultimately causing reconstruction failure. Since PoT does not use the symmetric masking, it succeeds in verification even for the invalid path, as shown in the lower part of Fig. 2.

B. Segment Routing over IPv6 Data Plane

SRv6 is a kind of source routing, which can realize SFC [4], [5]. Nodes in the service path $(v_0, \ldots, v_k)$, excluding the ingress node v_0, can be represented as an *SR policy*. There are three types of nodes in an SRv6 domain: *SR source node*, *SR segment endpoint node*, and *transit node*. The SR source node encapsulates an incoming packet with an *SR header* (SRH). SRH includes *segment list* and *segments left* fields defined by an SR policy. The segment list field contains m $(m \geq 1)$ *segment identifiers* (SIDs), each representing the IPv6 address of the node to be routed, stored in reverse order. The segments left field acts as a pointer of the SID of the next node. SRH can also include additional information as a *type length value* (TLV) object. In the proposed scheme, the aforementioned PoT metadata is included in the TLV object. (Details of the PoT metadata are given in Section IV-C.)

The node receiving an IPv6 packet acts as either an SR segment endpoint node or transit node. The SR segment endpoint node is a node whose SID matches the destination address of the outer IPv6 header. If the segments field of the received SRv6 packet does not have a zero value, the SR segment endpoint node inspects SRH of the received packet and updates the segments left and the next SID in the segment list with the destination address in the outer IPv6 header. Afterward, it forwards the packet to the next node according to the forwarding table. Otherwise, the SR segment endpoint node decapsulates the SRv6 packet and forwards the decapsulated packet to the destination node. The transit node is a node whose SID does not match the destination address of the outer IPv6 header and only forwards a packet to the next node based on IPv6 routing.

A 128-bit SRv6 SID is divided into three fields (i.e., *Locator*, *Function*, and *Argument*) and represented as `LOC:FUNCT:ARG`. Typically, the locator, function, and argument fields are defined with 48, 16, and 64 bits, respectively. The locator field (`LOC`) corresponds to an IPv6 network prefix used in routing protocols to identify a node hosting a specific function. The function field (`FUNCT`) designates the function to be executed at the node, while the argument field (`ARG`) optionally provides arguments to the function. The locator field routes the packet to a specific node, where a function identified by the function field is executed. Generally, the argument field is unused. Thus, the locator field supplies a 48-bit IPv6 network prefix (i.e., *Locator Block*) in an SRv6 network, with each node assigned a 64-bit IPv6 network sub-prefix that includes the locator block. This design supports a maximum of 2^{16} nodes in an SRv6 network.

Since the SID field in SRv6 is 128 bits (16 bytes), SRv6 routing requires additional $16m$ bytes to carry an SRv6 packet

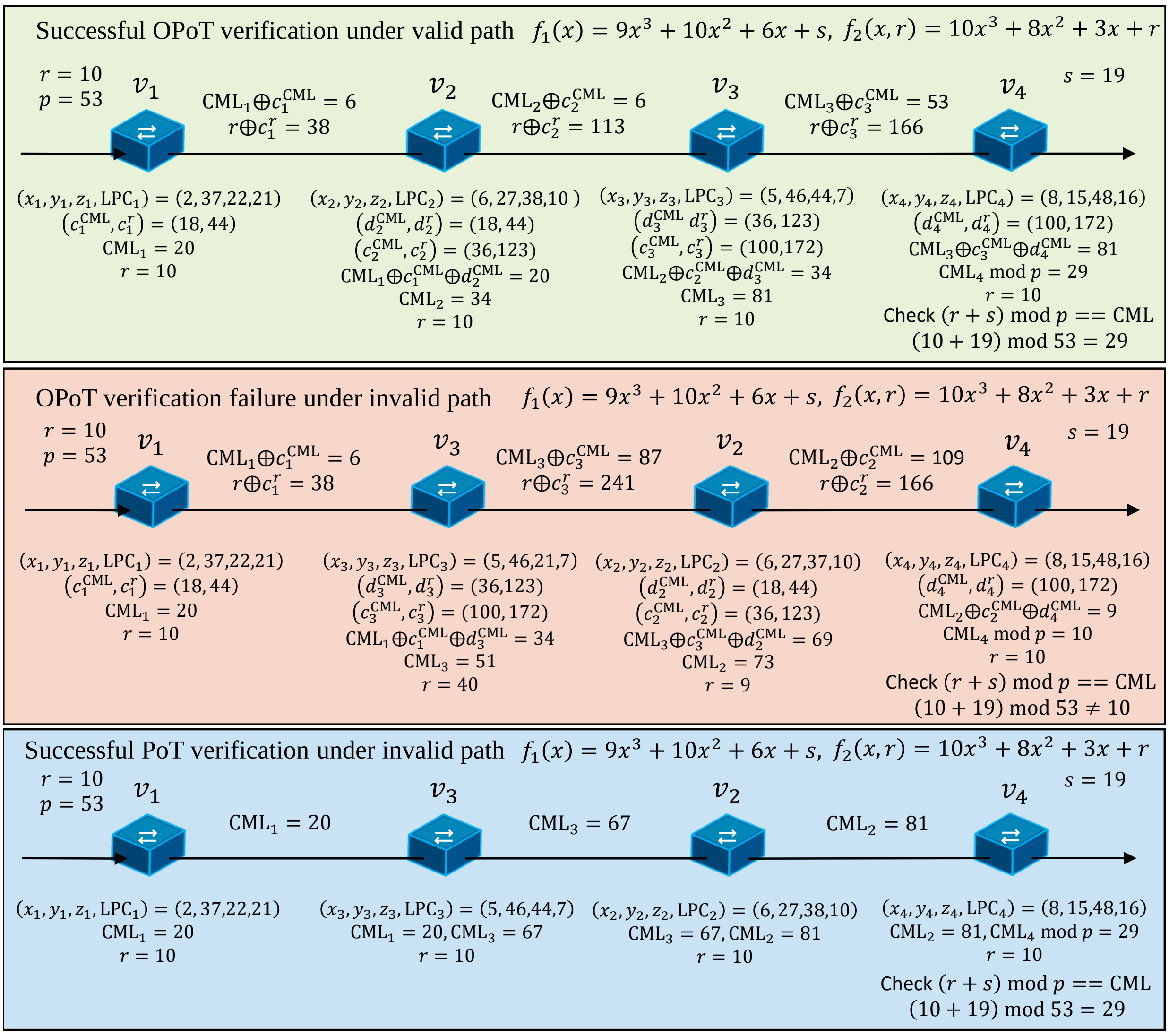

Successful OPoT verification under valid path $f_1(x) = 9x^3 + 10x^2 + 6x + s,\ f_2(x,r) = 10x^3 + 8x^2 + 3x + r$
$r = 10$
$p = 53$
v_1
$\mathrm{CML}_1 \oplus c_1^{\mathrm{CML}} = 6$
$r \oplus c_1^r = 38$
v_2
$\mathrm{CML}_2 \oplus c_2^{\mathrm{CML}} = 6$
$r \oplus c_2^r = 113$
v_3
$\mathrm{CML}_3 \oplus c_3^{\mathrm{CML}} = 53$
$r \oplus c_3^r = 166$
v_4
$s = 19$
$(x_1, y_1, z_1, \mathrm{LPC}_1) = (2, 37, 22, 21)$
$(c_1^{\mathrm{CML}}, c_1^r) = (18, 44)$
$\mathrm{CML}_1 = 20$
$r = 10$
$(x_2, y_2, z_2, \mathrm{LPC}_2) = (6, 27, 38, 10)$
$(d_2^{\mathrm{CML}}, d_2^r) = (18, 44)$
$(c_2^{\mathrm{CML}}, c_2^r) = (36, 123)$
$\mathrm{CML}_1 \oplus c_1^{\mathrm{CML}} \oplus d_2^{\mathrm{CML}} = 20$
$\mathrm{CML}_2 = 34$
$r = 10$
$(x_3, y_3, z_3, \mathrm{LPC}_3) = (5, 46, 44, 7)$
$(d_3^{\mathrm{CML}}, d_3^r) = (36, 123)$
$(c_3^{\mathrm{CML}}, c_3^r) = (100, 172)$
$\mathrm{CML}_2 \oplus c_2^{\mathrm{CML}} \oplus d_3^{\mathrm{CML}} = 34$
$\mathrm{CML}_3 = 81$
$r = 10$
$(x_4, y_4, z_4, \mathrm{LPC}_4) = (8, 15, 48, 16)$
$(d_4^{\mathrm{CML}}, d_4^r) = (100, 172)$
$\mathrm{CML}_3 \oplus c_3^{\mathrm{CML}} \oplus d_4^{\mathrm{CML}} = 81$
$\mathrm{CML}_4 \bmod p = 29$
$r = 10$
Check $(r + s) \bmod p == \mathrm{CML}$
$(10 + 19) \bmod 53 = 29$
OPoT verification failure under invalid path $f_1(x) = 9x^3 + 10x^2 + 6x + s,\ f_2(x,r) = 10x^3 + 8x^2 + 3x + r$
$r = 10$
$p = 53$
v_1
$\mathrm{CML}_1 \oplus c_1^{\mathrm{CML}} = 6$
$r \oplus c_1^r = 38$
v_3
$\mathrm{CML}_3 \oplus c_3^{\mathrm{CML}} = 87$
$r \oplus c_3^r = 241$
v_2
$\mathrm{CML}_2 \oplus c_2^{\mathrm{CML}} = 109$
$r \oplus c_2^r = 166$
v_4
$s = 19$
$(x_1, y_1, z_1, \mathrm{LPC}_1) = (2, 37, 22, 21)$
$(c_1^{\mathrm{CML}}, c_1^r) = (18, 44)$
$\mathrm{CML}_1 = 20$
$r = 10$
$(x_3, y_3, z_3, \mathrm{LPC}_3) = (5, 46, 21, 7)$
$(d_3^{\mathrm{CML}}, d_3^r) = (36, 123)$
$(c_3^{\mathrm{CML}}, c_3^r) = (100, 172)$
$\mathrm{CML}_1 \oplus c_1^{\mathrm{CML}} \oplus d_3^{\mathrm{CML}} = 34$
$\mathrm{CML}_3 = 51$
$r = 40$
$(x_2, y_2, z_2, \mathrm{LPC}_2) = (6, 27, 37, 10)$
$(d_2^{\mathrm{CML}}, d_2^r) = (18, 44)$
$(c_2^{\mathrm{CML}}, c_2^r) = (36, 123)$
$\mathrm{CML}_3 \oplus c_3^{\mathrm{CML}} \oplus d_2^{\mathrm{CML}} = 69$
$\mathrm{CML}_2 = 73$
$r = 9$
$(x_4, y_4, z_4, \mathrm{LPC}_4) = (8, 15, 48, 16)$
$(d_4^{\mathrm{CML}}, d_4^r) = (100, 172)$
$\mathrm{CML}_2 \oplus c_2^{\mathrm{CML}} \oplus d_4^{\mathrm{CML}} = 9$
$\mathrm{CML}_4 \bmod p = 10$
$r = 10$
Check $(r + s) \bmod p == \mathrm{CML}$
$(10 + 19) \bmod 53 \neq 10$
Successful PoT verification under invalid path $f_1(x) = 9x^3 + 10x^2 + 6x + s,\ f_2(x,r) = 10x^3 + 8x^2 + 3x + r$
$r = 10$
$p = 53$
v_1
$\mathrm{CML}_1 = 20$
v_3
$\mathrm{CML}_3 = 67$
v_2
$\mathrm{CML}_2 = 81$
v_4
$s = 19$
$(x_1, y_1, z_1, \mathrm{LPC}_1) = (2, 37, 22, 21)$
$\mathrm{CML}_1 = 20$
$r = 10$
$(x_3, y_3, z_3, \mathrm{LPC}_3) = (5, 46, 44, 7)$
$\mathrm{CML}_1 = 20, \mathrm{CML}_3 = 67$
$r = 10$
$(x_2, y_2, z_2, \mathrm{LPC}_2) = (6, 27, 38, 10)$
$\mathrm{CML}_3 = 67, \mathrm{CML}_2 = 81$
$r = 10$
$(x_4, y_4, z_4, \mathrm{LPC}_4) = (8, 15, 48, 16)$
$\mathrm{CML}_2 = 81, \mathrm{CML}_4 \bmod p = 29$
$r = 10$
Check $(r + s) \bmod p == \mathrm{CML}$
$(10 + 19) \bmod 53 = 29$

Fig. 2: OPoT reconstruction process.

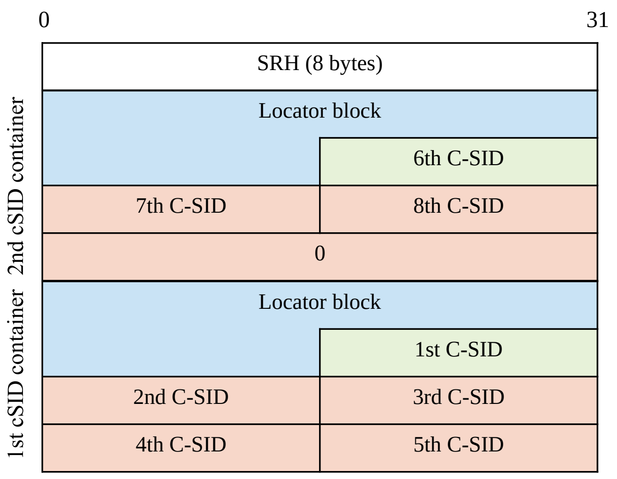


Fig. 3: cSID structure in compressed segment list ($m = 8$).

with a segment list of length m. To reduce the segment list length, compressed-SID (cSID) (a.k.a. *micro Segment Identifier* (uSID)) has been proposed in [6], [7]. cSIDs are encoded via either the *NEXT-C-SID* or *REPLACED-C-SID* encoding rule; we primarily focus on the NEXT-C-SID encoding rule. Fig. 3 illustrates the cSID structure in a compressed segment list ($m = 8$). A compressed segment list has one or more cSID containers, each structured in a 128-bit SID field (`LOC:FUNCT:ARG`) and stored in reverse order, as shown in Fig. 3. Each cSID container includes a 48-bit locator-block field (blue), a 16-bit locator-node and function field (green), and a 64-bit argument field (red). The locator-node and function field denotes the first cSID, and the argument field specifies the remaining cSIDs. This allows a single SID to represent up to five cSIDs, each occupying 2 bytes. The sixth and subsequent cSIDs are defined in the next cSID container. If a cSID container has fewer than five CSIDs, the remaining bits are zero-padded, indicating the end of a cSID container.

SRv6 with cSIDs operates as follows. An SR source node encapsulates an incoming packet with an SRH, as shown in Fig. 3, according to the SR policy and sets the first cSID container as the destination address of the outer IPv6 header.

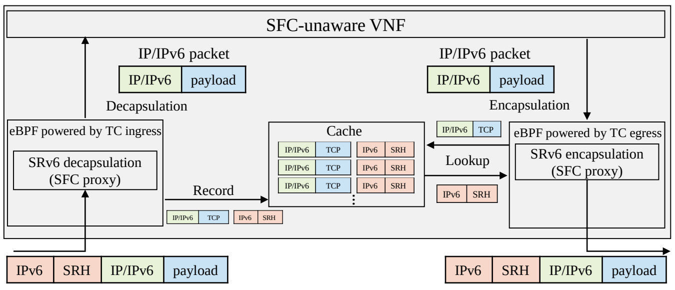


Fig. 4: Overview of eBPF-based SFC proxy.

Upon receiving an SRv6 packet, an SR segment endpoint node inspects the destination address of the outer IPv6 header to check if the argument field is non-zero. If the argument field is non-zero, the SR segment endpoint node updates the destination address to designate the next cSID by shifting the augment field left by 2 bytes and filling the last 2 bytes of the augment field with zeros. Otherwise, it decrements the segment left field and replaces the destination address with the next cSID container indicated by the segment left field. After completing these processes, the SR segment endpoint node forwards the packet to the next node with the updated destination address.

C. Extended Berkeley Packet Filter

eBPF is a technology to control the Linux kernel by injecting programs verified by an eBPF verifier into the kernel [18], [27]. The eBPF verifier assesses the stability and safety of programs, including absence of infinite loops and memory access violations, before loading them into the Linux kernel. An eBPF program can handle incoming (resp. outgoing) packets to (resp. from) a network interface by integrating with *traffic control* (TC) ingress (resp. egress). eBPF is utilized in various networking domains, including packet filtering [28], SFC proxy [20], SRv6 [29], and VNFs [30], [31]. A *BPF map* serves as a key–value store, facilitating data sharing between kernel space and user space.

D. eBPF-based SFC Proxy

An SFC proxy is a technology that integrates SFC-unaware VNFs into SFC. In [20], the eBPF-based SFC proxy is implemented, which decapsulates an SFC-related (encapsulated) packet before processing an SFC-unaware VNF and encapsulates it after packet processing using a BPF map. Fig. 4 illustrates the overview of the eBPF-based SFC proxy. The SFC proxy operates within the TC ingress and egress. Upon the arrival of a newly encapsulated packet (i.e., an SRv6 packet) at the network interface, the eBPF program attached at the TC ingress is activated. It decapsulates the SRv6 packet, stores a key-value pair in the BPF map, and forwards the decapsulated SRv6 packet to the user space program and/or another eBPF program. Since this key-value pair will be used in the later packet encapsulation process, key collisions between the packet being processed and newly arriving packets must be avoided. Therefore, the SFC proxy supports only protocols that contain unique header information to identify individual packets (e.g., TCP). As in [20], the unique key is composed of the source and destination IP addresses, the identification field of IP header, the TCP source and destination port numbers, and the TCP sequence and acknowledgment numbers from the inner header, while the outer IPv6 header and SRH form the corresponding value. The SFC-unaware VNF can handle the decapsulated packet and forwards it to the next node. Upon the departure of the processed packet, the eBPF program attached at the TC egress is activated. This program running on the TC egress encapsulates the processed packet by retrieving the outer IPv6 header and SRH from the BPF map using the IPv6 header of the processed packet as a key.

IV. eBPF-BASED OPoT FOR PATH VERIFICATION

A. Overview

Fig. 5 illustrates the overview of the proposed scheme to implement both OPoT and SFC proxy using eBPF. The OPoT function comprises three types of functions operating within the TC ingress of the ingress node (Fig. 5a), TC egress of egress node (Fig. 5b), and TC egress of the endpoint node (Figs. 5c and 5d), respectively. The SR source node is responsible for the ingress node. If the segments left field of the received SRv6 packet is zero, the SR segment endpoint node is responsible for the egress node; otherwise, it is responsible for the endpoint node. The SFC proxy function comprises two types of functions, which are attached to the TC ingress and egress of the endpoint nodes (Fig. 5c).

The eBPF program attached to the TC ingress of the ingress node initializes the TLV option (i.e., PoT metadata) and performs SRH encapsulation. (See the detail of PoT metadata in Section IV-C.) In case of the SFC-unaware VNF, the eBPF program attached to the TC ingress of the endpoint node records the header information of the incoming packet to a cache and decapsulates the incoming packet. Note that the cache is implemented by the BPF map, as described in Section IV-B. After packet processing by VNF, the eBPF program attached to the TC egress of the endpoint node retrieves the header information from the cache and encapsulates the processed packet. Subsequently the eBPF program performs the OPoT reconstruction algorithm. On the other hand, in case of the SFC-aware VNF, the eBPF program only executes the OPoT reconstruction algorithm at the TC egress of the endpoint node. The eBPF program attached to the TC ingress of the egress node is responsible for the OPoT-based path verification and SRv6 decapsulation. (The details of the OPoT reconstruction algorithm and SFC proxy will be presented in Section IV-D.)

B. BPF Maps

Table II presents the BPF maps used in the proposed scheme. Ingress, endpoint, and egress nodes $\forall v_j \in \pi_l$ maintain three types of information (SR policies, OPoT parameters, and SFC proxies) using the BPF maps, respectively. The SR policy map stores both the prefix length and destination address as a key, with the SR policy as its corresponding value. Let $\mathcal{L}$ denote a set of SR policies managed by the SDN controller.

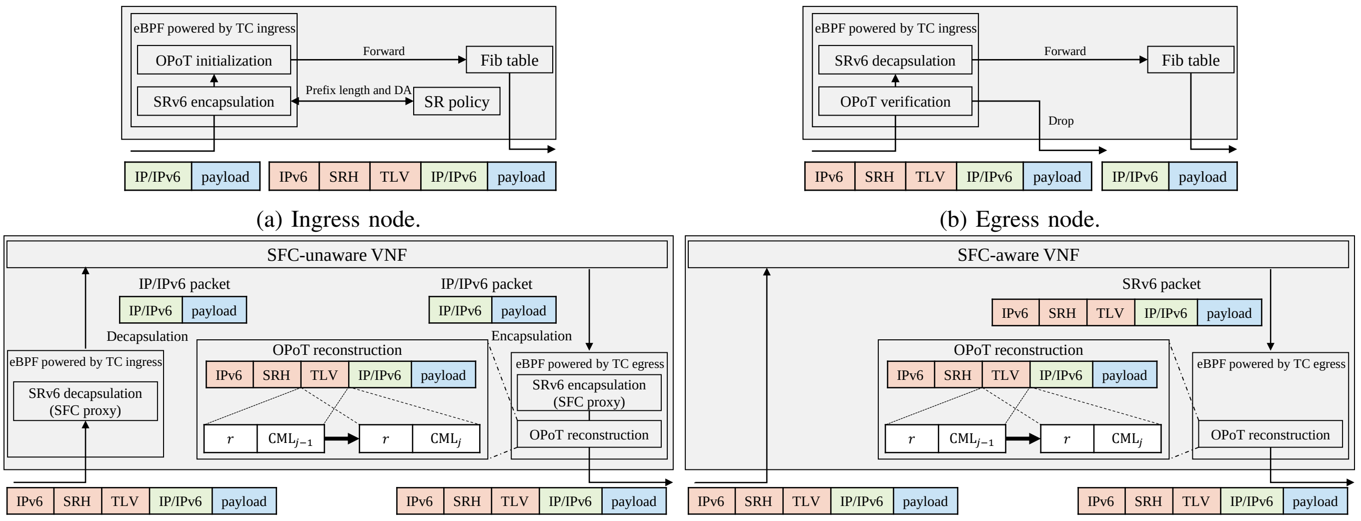


Fig. 5: Overview of proposed scheme.

TABLE II: BPF maps.

Name	Key	Value
SR policy map	Prefix length and destination address	SR policy l
OPoT map	Prefix length and destination address	OPoT parameters $\mathrm{P}_{l,j}$
SFC proxy map	Source and destination IP addresses, identification field of IP header, TCP source and destination port numbers, and TCP sequence and acknowledgment numbers from the inner packet	Outer IPv6 header and SRH

The SDN controller records the SR policy $l \in \mathcal{L}$ into the SR policy map managed by both ingress node v_0 and nodes $(v_1, \dots, v_{k-1})$ included in the segment list of the SR policy l.

The OPoT map manages parameters used for OPoT. Let the 8-tuple OPoT parameter of node $v_j \in \pi_l$ under policy l be defined as $\mathrm{P}_{l,j} = \langle x_{l,j}, y_{l,j}, \boldsymbol{b}_l, \mathrm{LPC}_{l,j}, p, \boldsymbol{c}_{l,j}, \boldsymbol{d}_{l,j}, s_l \rangle$. The sizes of parameters $x_{l,j}$, $y_{l,j}$, $\mathrm{LPC}_{l,j}$, p, and s_l are assumed to be 8 bytes each. The size of the coefficient vector $\boldsymbol{b}_l$ is $8(|\pi_l| - 1)$ bytes. The cipher key $\boldsymbol{c}_{l,j}$ and decipher key $\boldsymbol{d}_{l,j}$ are each 16 bytes. Therefore, the total OPoT parameter size is $64 + 8|\pi_l|$ bytes. Each element is prepared by the SDN controller per SR policy l according to the process mentioned in Section III-A and is stored in the OPoT map at node v_j as $\mathrm{P}_{l,j}$. The eBPF program running on the TC ingress of the endpoint node records the key-value pair, as defined in Section III-D, into the SFC proxy map. It is assumed that these BPF maps are updated and/or deleted by the SDN controller.

C. Metadata of OPoT

Fig. 6 illustrates the TLV of OPoT, which is required to implement OPoT in the SRv6 domain. As shown in Fig. 6, the TLV of OPoT contains the following fields: type (1 byte), length (1 byte), random value (8 byte), and cumulative value fields (8 byte). Since the SRH size is defined as a multiple of 8 octets [32], the TLV size must adhere to this rule; therefore 6-byte zero padding is required between the length and random value fields. The random value r is generated by the ingress node (i.e., v_0) in the SRv6 domain and is assigned to the random value field. The cumulative value field is computed by the node v_j in the SRv6 domain according to Eq. (4).

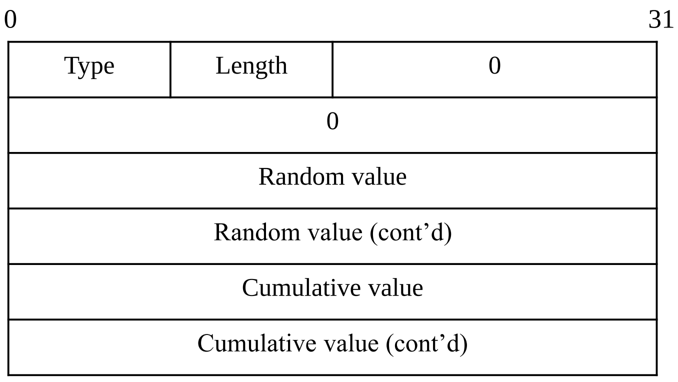


Fig. 6: TLV of OPoT.

D. eBPF-based Implementation of OPoT Reconstruction

Pseudo-code 1 presents the eBPF-based Implementation of OPoT reconstruction. The pseudocode differs for ingress, endpoint, and egress nodes, respectively. At the ingress (resp. egress) node, the eBPF program, the OPoT-Initialization (resp. OPoT-Verification) function, is attached to the TC ingress, as depicted in Fig. 5a (resp. Fig. 5b), while at the endpoint node, the OPoT-Reconstruction function is

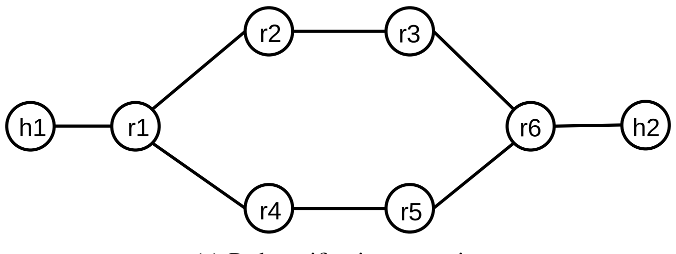
Pseudo-code 1 eBPF-based Implementation of OPoT reconstruction.

```
1:  function OPoT-INITIALIZATION($pkt$)
2:      $l \leftarrow$ GET_POLICY($pkt$)
3:      $r \leftarrow$ GET_RANDOM_VALUE()
4:      CML $\leftarrow 0$
5:      $pkt \leftarrow$ ENCAPSULATE($pkt, l$).
6:      $P_{l,j} \leftarrow$ GET_OPOT_PARAMETERS($pkt$).
7:      $pkt \leftarrow$ UPDATE_TLV_OPTION($pkt, P_{l,j}$)
8:      FORWARD($pkt, table$)
9:  function SFC-PROXY($pkt$)
10:     $inner, outer \leftarrow$ GET_HEADERS($pkt$)
11:     RECORD_INTO_SFC_PROXY_MAP($inner, outer$)
12:     $pkt \leftarrow$ DECAPSULATE($pkt$)
13:     PASS_TO_VNF($pkt$)
14: function OPoT-RECONSTRUCTION($pkt$)
15:     if VNF running on $v_j$ is SFC-unaware then
16:         $inner \leftarrow$ GET_HEADER($pkt$)
17:         $outer \leftarrow$ LOOKUP_SFC_PROXY_MAP($inner$)
18:         $pkt \leftarrow$ ENCAPSULATE($pkt, outer$).
19:     ($r$, CML) $\leftarrow$ GET_TLV_OPTION($pkt$)
20:     $P_{l,j} \leftarrow$ GET_OPOT_PARAMETERS($pkt$).
21:     $pkt \leftarrow$ UPDATE_TLV_OPTION($pkt, P_{l,j}$)
22:     FORWARD($pkt, table$)
23: function OPoT-VERIFICATION($pkt$)
24:     $P_{l,j} \leftarrow$ GET_OPOT_PARAMETERS($pkt$).
25:     if VERIFY($pkt, s_l$) is successful then
26:         $pkt \leftarrow$ DECAPSULATE($pkt$)
27:         FORWARD($pkt, table$)
28:     else
29:         DROP($pkt$)
```

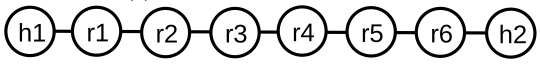
attached to the TC egress, as shown in Figs. 5c and 5d. If the endpoint node does not support SFC-aware VNFs, the SFC-PROXY function is further attached to the TC ingress in Fig 5c, in addition to the TC egress.

Upon the arrival of a new packet pkt at the TC ingress of ingress node v_j, the eBPF program (i.e., the INITIALIZATION function) is activated (lines 1–8). It first retrieves the SR policy l from the SR policy map using the prefix length and destination address of packet pkt as a key (line 2), generates a random value r, and initializes CML with zero (lines 3 and 4). It then encapsulates the packet pkt according to the SR policy l (line 5). Next, it retrieves the OPoT parameters $P_{l,j}$ from the OPoT map using the prefix length and destination address of packet pkt as a key (line 6) and updates CML in the TLV option according to Eq. (4) (line 7). Finally, it forwards the packet pkt to the next node v_{j+1} according to the forwarding information base (FIB) table $table$ (line 8).

Upon the arrival of a new SRv6 packet at the TC ingress of the endpoint node v_j, the eBPF program (i.e., the SFC-PROXY function) is activated if the endpoint node v_j only supports an SFC-unaware VNF (lines 9–13). Otherwise, it is not activated, and the packet pkt is directly processed by the SFC-aware VNF. In the SFC-PROXY() function, the eBPF program records a key–value pair ($inner, outer$) into the SFC proxy map, where $inner$ is the inner IPv6 header of the SRv6 packet and $outer$ is the corresponding outer IPv6 header and SRH, and then decapsulates it (lines 10–12). Afterwards, the decapsulated packet pkt is processed by the SFC-unaware VNF (line 13).



(a) Path verification scenario.



(b) Performance limitation scenario.

Fig. 7: Evaluation network.

After the packet pkt is processed by the VNF, it will be forwarded to the next node v_{j+1}. At this moment, the eBPF program (i.e., the RECONSTRUCTION function) is activated at the TC egress of the endpoint node v_j (lines 14–22). If the VNF running on the endpoint node v_j is SFC-unaware, the eBPF program first retrieves $outer$ (i.e., the outer IPv6 header and SRH) from the SFC proxy map using the inner IP/IPv6 header $inner$ of the processed packet pkt as a key. It then encapsulates the packet pkt according to the outer IPv6 header and SRH (lines 15–18). Next, the eBPF program retrieves the random value r and the cumulative value CML from the TLV option of the processed packet pkt (line 19). It also retrieves the OPoT parameters $P_{l,j}$ from the OPoT map using the prefix length and destination address of the packet pkt as a key (line 20). It updates CML in the TLV option according to Eq. (4) (line 21). Finally, it forwards the packet pkt to the next node v_{j+1} according to the FIB table $table$ (line 22).

Upon the arrival of an SRv6 packet at the egress node, the eBPF program (i.e., the VERIFICATION function) is activated at the TC ingress (lines 23–29). It verifies secret s using random value r and cumulative value CML (line 25). If the verification is successful, the eBPF program decapsulates packet pkt and forwards it according to FIB Table $table$ (lines 26–27). Otherwise, the eBPF program drops packet pkt (line 29).

V. EVALUATION

A. Path Verification of OPoT

1) Evaluation Scenario: In the evaluation, we utilize a server featuring the 13th Gen Intel® Core™ i7-13700K CPU, 64 GB memory, and Ubuntu 22.04 LTS (kernel version: 6.2.0). We emulate the network depicted in Fig. 7a on the server using the kernel virtual machine (KVM) [33]. Each node on the virtual network is provisioned with 1 vCPU, 4 GB memory, and Ubuntu 22.04 LTS (kernel version: 6.2.0). Since the prototype of the proposed scheme does not include a

TABLE III: Scheme comparison.

Scheme	Protocol	eBPF	OPoT	SFC proxy	MSS
IPv6	IPv6	-	-	-	1440
SRv6	SRv6	-	-	-	1312
SRv6$^{\mathrm{cSID}}$	SRv6 with cSID	-	-	-	1376
OPoT	SRv6	✓	✓	-	1288
OPoT$_{\mathrm{proxy}}$	SRv6	✓	✓	✓	1288
OPoT$^{\mathrm{cSID}}$	SRv6 with cSID	✓	✓	-	1352
OPoT$^{\mathrm{cSID}}_{\mathrm{proxy}}$	SRv6 with cSID	✓	✓	✓	1352

control plane, we manually configure the parameters of each node using a configuration file. Suppose that end-host `h1` tries to communicate with end-host `h2` through an SR domain composed of nodes from `r1` to `r6`. Node `r1` serves as the ingress node of SRv6, responsible for encapsulating incoming packets from end-host `h1`. Nodes `r2`–`r5` function as endpoint nodes of SRv6, equipped with VNFs solely for forwarding, responsible for forwarding incoming packets based on the forwarding table. Node `r6` acts as the egress node of SRv6, tasked with decapsulating incoming packets and forwarding them to end-host `h2`.

For comparative analysis, we prepare seven schemes: IPv6, SRv6, OPoT (eBPF-based OPoT), OPoT$_{\mathrm{proxy}}$ (eBPF-based OPoT with SFC proxy), SRv6$^{\mathrm{cSID}}$ (SRv6 with cSIDs), OPoT$^{\mathrm{cSID}}$ (eBPF-based OPoT with cSIDs), and OPoT$^{\mathrm{cSID}}_{\mathrm{proxy}}$ (eBPF-based OPoT with cSIDs and SFC proxy), as shown in Table III. In the IPv6 scheme, all nodes follow the normal IPv6 protocol, where there is no SRv6 domain in the network. In the SRv6 scheme, we adopt the standard SRv6 encapsulation/decapsulation implementation in the Linux kernel and deploy it in nodes $\texttt{r1}, \ldots, \texttt{r6}$. In the OPoT scheme, we implement SRv6 encapsulation/decapsulation and OPoT as an eBPF program. The OPoT$^{\mathrm{cSID}}$ scheme adopts cSIDs instead of original SIDs. In addition to these eBPF programs, the OPoT$_{\mathrm{proxy}}$ and OPoT$^{\mathrm{cSID}}_{\mathrm{proxy}}$ schemes also implement an eBPF-based SFC proxy. In these OPoT-based schemes, the eBPF programs are deployed to part of nodes: `r1`, `r2`, `r3`, and `r6`, which results in the segment list of $(\texttt{r2}, \texttt{r3}, \texttt{r6})$.

We define the following evaluation scenario: Firstly, end-host `h1` transmits an ICMP packet three times to end-host `h2` using the *ping* command. Secondly, the segment list registered in the SR policy map of `r1` is modified to $(\texttt{r4}, \texttt{r5}, \texttt{r6})$ due to misconfiguration or tampering. Finally, `h1` sends an ICMP packet three times to `h2` again.

2) Verification Result: Fig. 8 illustrates path verification results by the eBPF-based OPoT. By focusing on the highlighted green area in Fig. 8, we confirm that OPoT successfully verifies that the first three packets traverse the nodes along the designated path ($\texttt{r1} \rightarrow \texttt{r2} \rightarrow \texttt{r3} \rightarrow \texttt{r6}$). Consequently, the eBPF program at node `r6` responds with “Verification successful” and forwards the packets to `h2`. On the other hand, when the segment list is modified, the eBPF program at node `r6` responds with “Verification failed” and drops the packet. This indicates that the last three packets follow the altered path ($\texttt{r1} \rightarrow \texttt{r4} \rightarrow \texttt{r5} \rightarrow \texttt{r6}$), as shown in the highlighted red area of Fig. 8. We also confirm that the eBPF-based OPoT with SFC proxy scheme shows the same behavior as the eBPF-based OPoT scheme.

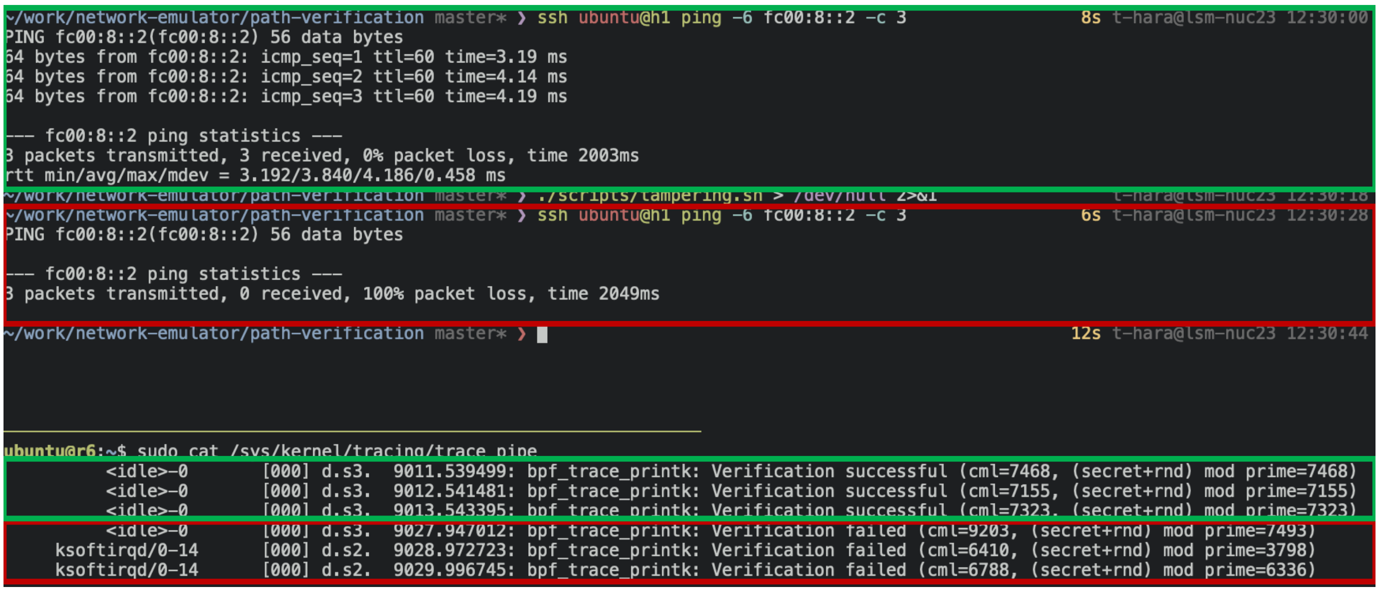
B. Performance Limitation

1) Evaluation Scenario: In this evaluation, we measure both the packet rate and bit rate. We emulate the network depicted in Fig. 7b on the server using the same settings, as in Section V-A1. The maximum bandwidth of each node is limited to a specific value between from 1 Gbps and 5 Gbps, with a maximum transmission unit (MTU) of 1500 bytes for each node interface, by editing the corresponding configuration file in the network XML format [34]. Libvirt enforces bandwidth limits using TC with the *hierarchical token bucket* (HTB) queuing discipline. TCP packets are sent from `h1` to `h2` for 100 seconds using the *iperf3* command. To accommodate SRv6 encapsulation, we adjust the maximum segment size (MSS) of TCP packets when sending them via *iperf3*. MSS for SRv6 (resp. OPoT) packets with the segment list length of m is defined as $1500 - 60 - (48 + 16m)$ bytes (resp. $1500 - 60 - (72 + 16m)$ bytes). Note that the IPv6 (resp. TCP) header size is set to 40 bytes (resp. 20 bytes). In schemes based on the original SRv6, the segment list length m is set to 5, whereas in schemes based on SRv6 with cSIDs, m is set to 1 (i.e., a compressed segment list contains five cSIDs.)

As for evaluation metrics, we consider both the packet rate and the bit rate. Since the MSS of TCP packets varies depending on the schemes used, we define the packet sending efficiencies $\hat{\rho}_{\mathrm{SRv6}}$, $\hat{\rho}^{\mathrm{cSID}}_{\mathrm{SRv6}}$, $\hat{\rho}_{\mathrm{OPoT}}$, $\hat{\rho}^{\mathrm{cSID}}_{\mathrm{OPoT}}$ as the ratios of the inner packet size of Srv6, SRv6$^{\mathrm{cSID}}$, OPoT, and OPoT$^{\mathrm{cSID}}$ to the IPv6 packet size. In addition, we define ρ as the ratio of the bit rate of each scheme to that of IPv6. To evaluate the overall processing overhead at the nodes along the path, we use the round-trip time (RTT). We send 100 ICMPv6 packets from `h1` to `h2` to measure RTT samples. It should be noted that the eBPF-based OPoT is applied only to packets directed to `h2`.

2) Throughput: In this section, we investigate the maximum link capacity at which actual transmission performance can nearly achieve the theoretical upper limit (i.e., packet sending efficiency). We also evaluate the number of bits that can be transmitted and processed by the eBPF-based OPoT implementation compared with IPv6, SRv6, and SRv6 with cSID transmission. Figs. 9 and 10 illustrate how the maximum link speed $B_{\max}$ affects the packet rate and the bit rate, respectively. We observe that all schemes exhibit a similar trend in both packet rate and bit rate, showing nearly identical packet rates when $B_{\max} \leq 3$, indicating that the link capacity is saturated. This suggests that all schemes can operate in real-time up to $B_{\max} \leq 3$, but their performance differs when $B_{\max} \geq 4$. At $B_{\max} = 5$ Gbps, the packet rates of IPv6, SRv6, OPoT, OPoT$_{\mathrm{proxy}}$, SRv6$^{\mathrm{cSID}}$, OPoT$^{\mathrm{cSID}}$, and OPoT$^{\mathrm{cSID}}_{\mathrm{proxy}}$ schemes achieve 0.379 Mpps, 0.374 Mpps, 0.350 Mpps, 0.343 Mpps,

SUBMITTED VERSION



```
~/work/network-emulator/path-verification master* ❯ ssh ubuntu@h1 ping -6 fc00:8::2 -c 3            8s t-hara@lsm-nuc23 12:30:00
PING fc00:8::2(fc00:8::2) 56 data bytes
64 bytes from fc00:8::2: icmp_seq=1 ttl=60 time=3.19 ms
64 bytes from fc00:8::2: icmp_seq=2 ttl=60 time=4.14 ms
64 bytes from fc00:8::2: icmp_seq=3 ttl=60 time=4.19 ms

--- fc00:8::2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 3.192/3.840/4.186/0.458 ms
~/work/network-emulator/path-verification master* ❯ ./scripts/tampering.sh > /dev/null 2>&1            t-hara@lsm-nuc23 12:30:18
~/work/network-emulator/path-verification master* ❯ ssh ubuntu@h1 ping -6 fc00:8::2 -c 3            6s t-hara@lsm-nuc23 12:30:28
PING fc00:8::2(fc00:8::2) 56 data bytes

--- fc00:8::2 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2049ms

~/work/network-emulator/path-verification master* ❯                                                 12s t-hara@lsm-nuc23 12:30:44




ubuntu@r6:~$ sudo cat /sys/kernel/tracing/trace_pipe
          <idle>-0       [000] d.s3.  9011.539499: bpf_trace_printk: Verification successful (cml=7468, (secret+rnd) mod prime=7468)
          <idle>-0       [000] d.s3.  9012.541481: bpf_trace_printk: Verification successful (cml=7155, (secret+rnd) mod prime=7155)
          <idle>-0       [000] d.s3.  9013.543395: bpf_trace_printk: Verification successful (cml=7323, (secret+rnd) mod prime=7323)
          <idle>-0       [000] d.s3.  9027.947012: bpf_trace_printk: Verification failed (cml=9203, (secret+rnd) mod prime=7493)
     ksoftirqd/0-14      [000] d.s2.  9028.972723: bpf_trace_printk: Verification failed (cml=6410, (secret+rnd) mod prime=3798)
     ksoftirqd/0-14      [000] d.s2.  9029.996745: bpf_trace_printk: Verification failed (cml=6788, (secret+rnd) mod prime=6336)
```

Fig. 8: Path verification results by eBPF-based OPoT.

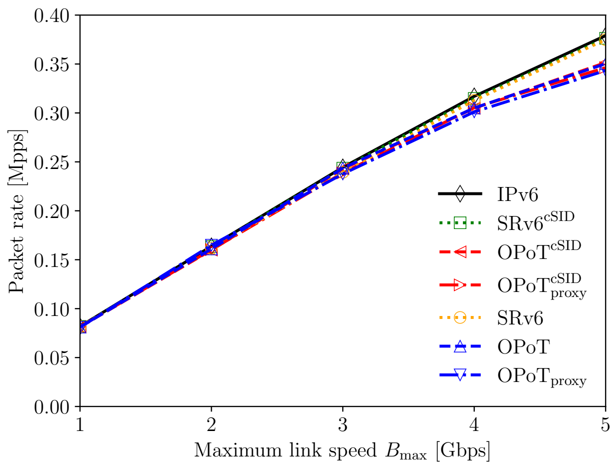


Fig. 9: Relationship between the maximum link speed $B_{\max}$ and the packet rate.

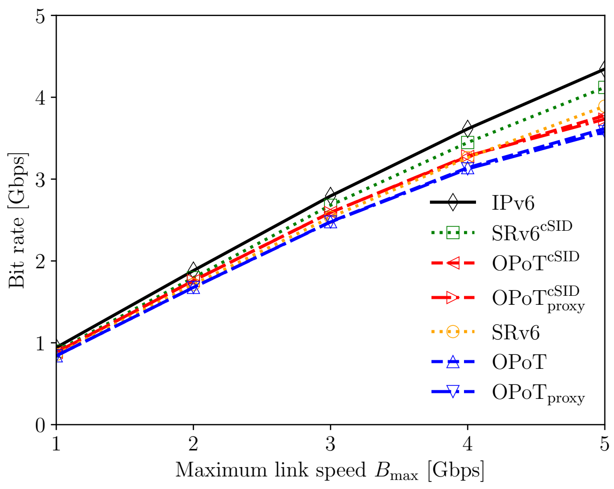


Fig. 10: Relationship between the maximum link speed and the bit rate.

0.376 Mpps, 0.351 Mpps, and 0.346 Mpps, respectively. Correspondingly, their bit rates reach 4.35 Gbps, 3.89 Gbps, 3.62 Gbps, 3.58 Gbps, 4.12 Gbps, 3.78 Gbps, and 3.73 Gbps, respectively. Specifically, the performance degradation of OPoT, OPoT$_{\mathrm{proxy}}$, OPoT$^{\mathrm{cSID}}$, and OPoT$^{\mathrm{cSID}}_{\mathrm{proxy}}$ is evaluated to be 7.0%, 8.1%, 8.3%, and 9.4% at $B_{\max} = 5$ Gbps, compared with SRv6 and SRv6$^{\mathrm{cSID}}$ schemes, due to the additional processing load of the OPoT reconstruction algorithm and the SFC proxy.

The degradation in both packet rate and bit rate arises from two factors: (1) packet sending efficiency and (2) packet processing capability. To understand how these factors affect the bit rate degradation, we illustrate the relationship between the maximum link speed $B_{\max}$ and the bit rate ratio ρ compared with IPv6 transmission in Fig. 11. By focusing on $\hat{\rho}^{\mathrm{cSID}}_{\mathrm{SRv6}}$, $\hat{\rho}_{\mathrm{SRv6}}$, $\hat{\rho}_{\mathrm{OPoT}}$, and $\hat{\rho}^{\mathrm{cSID}}_{\mathrm{OPoT}}$, we confirm that the SRv6 transmission, required in SRv6$^{\mathrm{cSID}}$, SRv6, OPoT OPoT$_{\mathrm{proxy}}$, OPoT$^{\mathrm{cSID}}$, and OPoT$^{\mathrm{cSID}}_{\mathrm{proxy}}$ schemes, results in bit rate degradation compared to IPv6 transmission. This degradation is caused by the increased header size (i.e., decreased packet sending efficiency) due to the SRv6 encapsulation, as shown in Table III. Notably, schemes based on SRv6 with cSIDs attain higher bit rates than schemes based on the original SRv6 when $B_{\max} \leq 3$ due to smaller header sizes. Conversely, examining the gap between $\hat{\rho}$ and ρ for each scheme, we observe that this gap remains almost constant when $B_{\max} \leq 3$, but the bit rate degradation becomes more significant as $B_{\max}$ increases to 5 Gbps. Specifically, we observe that ρ of OPoT$^{\mathrm{cSID}}$ and SRv6 reverses when $B_{\max} = 5$ Gbps. Furthermore, OPoT$_{\mathrm{proxy}}$ and OPoT$^{\mathrm{cSID}}_{\mathrm{proxy}}$ exhibit slightly lower ρ than OPoT and OPoT$^{\mathrm{cSID}}$, respectively, when $B_{\max} = 5$, due to the impact of the SFC proxy. This result suggests the primary factor contributing to performance degradation is the packet processing capability at high maximum link speeds $B_{\max}$.

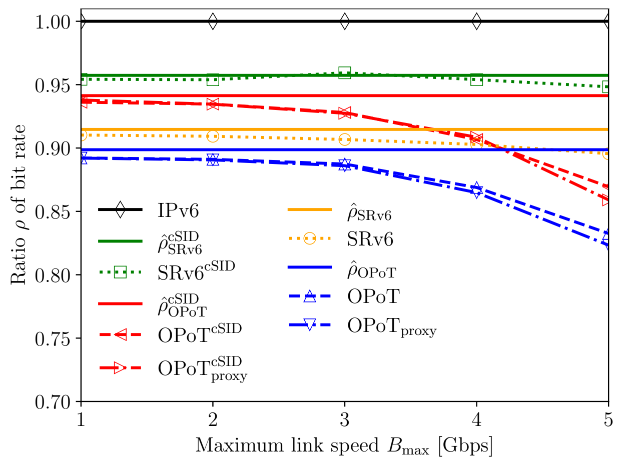


Fig. 11: Relationship between the maximum link speed and the ratio ρ of bit rate.

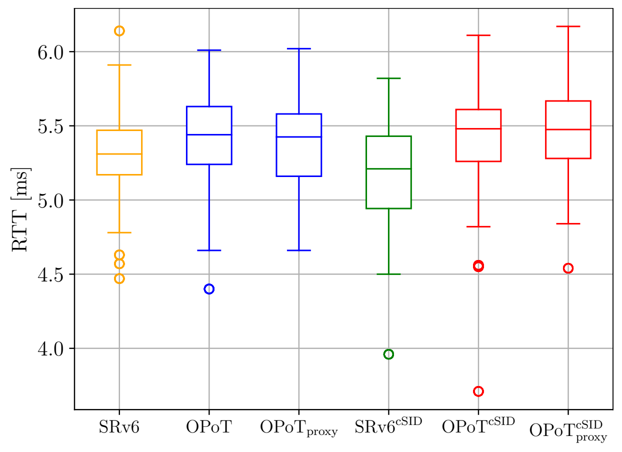


Fig. 12: End-to-end RTT of ICMPv6 packets.

3) End-to-End RTT Evaluation: In this section, we evaluate the end-to-end (E2E) RTT of ICMPv6 packets by comparing the six schemes: SRv6, OPoT, OPoT$_{\mathrm{proxy}}$, SRv6$^{\mathrm{cSID}}$, OPoT$^{\mathrm{cSID}}$, and OPoT$_{\mathrm{proxy}}^{\mathrm{cSID}}$. Fig. 12 illustrates the box-and-whisker plot of the E2E RTT for ICMPv6 packets. We observe that the average E2E RTT for SRv6, OPoT, OPoT$_{\mathrm{proxy}}$, SRv6$^{\mathrm{cSID}}$, OPoT$^{\mathrm{cSID}}$, and OPoT$_{\mathrm{proxy}}^{\mathrm{cSID}}$ shows 5.30 ms, 5.42 ms, 5.39 ms, 5.17 ms, 5.41 ms, and 5.47 ms, respectively. This slight increase in E2E RTT compared to SRv6 appears to result from the additional overhead introduced by OPoT and SFC proxy.

VI. Conclusion

In this paper, by leveraging extended Berkeley Packet Filter (eBPF), we have implemented ordered proof of transit (OPoT) in a lightweight way, achieving trustworthy service function chaining (SFC) with path verification capability in the segment routing over IPv6 data plane (SRv6) domain. We have further integrated it with the SFC proxy to achieve the trustworthy SFC even in environments with a mixture of SFC-aware and unaware virtual network functions (VNFs).

Through experiments, we have demonstrated that the proposed scheme enables the eBPF program to successfully verify the specified path without requiring modifications to the VNFs operating in the user space. Furthermore, we have identified the performance limitations of the proposed scheme in terms of bit rate, packet rate, and end-to-end round trip time. The results indicate that the proposed scheme nearly reaches the theoretical performance limit at a maximum link speed of up to 3 Gbps; however, the performance degrades at a maximum link speed of 4 Gbps or higher due to limitations in packet processing capability. These results suggest that the proposed scheme can accommodate various network services (e.g., IoT networks, traffic control for multi-access edge computing, and network security services). In future work, we aim to implement OPoT-based path verification on smart network interface cards (SmartNICs) to realize faster packet processing. We also plan to implement a general framework, In-situ Operations, Administration, and Maintenance (IOAM), in eBPF for network telemetry and develop a control plane using Open vSwitch (OVS) compatible with eBPF [35].

References

[1] J. M. Halpern and C. Pignataro, "Service Function Chaining (SFC) Architecture," RFC 7665, Oct. 2015. [Online]. Available: https://www.rfc-editor.org/info/rfc7665

[2] M. Sasabe and T. Hara, "Capacitated Shortest Path Tour Problem-Based Integer Linear Programming for Service Chaining and Function Placement in NFV Networks," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 104–117, Mar. 2021.

[3] A. Lara, A. Kolasani, and B. Ramamurthy, "Network Innovation using OpenFlow: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 493–512, 2014.

[4] C. Filsfils, P. Camarillo, J. Leddy, D. Voyer, S. Matsushima, and Z. Li, "Segment Routing over IPv6 (SRv6) Network Programming," RFC 8986, Feb. 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc8986

[5] P. L. Ventre, S. Salsano, M. Polverini, A. Cianfrani, A. Abdelsalam, C. Filsfils, P. Camarillo, and F. Clad, "Segment Routing: A Comprehensive Survey of Research Activities, Standardization Efforts, and Implementation Results," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 1, pp. 182–221, 2021.

[6] W. Cheng, C. Filsfils, Z. Li, B. Decraene, and F. Clad, "Compressed SRv6 Segment List Encoding," Internet Engineering Task Force, Internet-Draft draft-ietf-spring-srv6-srh-compression-19, Nov. 2024, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-spring-srv6-srh-compression/19/

[7] A. Tulumello, A. Mayer, M. Bonola, P. Lungaroni, C. Scarpitta, S. Salsano, A. Abdelsalam, P. Camarillo, D. Dukes, F. Clad, and C. Filsfils, "Micro SIDs: A Solution for Efficient Representation of Segment IDs in SRv6 Networks," vol. 20, no. 1, pp. 774–786.

[8] A. Bashandy, C. Filsfils, S. Previdi, B. Decraene, S. Litkowski, and R. Shakir, "Segment Routing with the MPLS Data Plane," RFC 8660, Dec. 2019. [Online]. Available: https://www.rfc-editor.org/info/rfc8660

[9] P. Quinn, U. Elzur, and C. Pignataro, "Network Service Header (NSH)," RFC 8300, Jan. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8300

[10] T. Sasaki, C. Pappas, T. Lee, T. Hoefler, and A. Perrig, "SDNsec: Forwarding Accountability for the SDN Data Plane," in *Proc. of International Conference on Computer Communication and Networks (ICCCN)*, Aug. 2016, pp. 1–10.

[11] P. Zhang, "Towards Rule Enforcement Verification for Software Defined Networks," in *Proc. of IEEE Conference on Computer Communications*, May 2017, pp. 1–9.

[12] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures," in *Proc. of 2015 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2015.

[13] M. Pattaranantakul, C. Vorakulpipat, and T. Takahashi, "Service Function Chaining Security Survey: Addressing Security Challenges and Threats," *Computer Networks*, vol. 221, p. 109484, Feb. 2023.

[14] F. Brockners, S. Bhandari, T. Mizrahi, S. Dara, and S. Youell, "Proof of Transit," Internet Engineering Task Force, Internet-Draft draft-ietf-sfc-proof-of-transit-08, Nov. 2020, Work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-sfc-proof-of-transit/08/

[15] F. Brockners, S. Bhandari, D. Bernier, and T. Mizrahi, “In Situ Operations, Administration, and Maintenance (IOAM) Deployment,” RFC 9378, Apr. 2023. [Online]. Available: https://www.rfc-editor.org/info/rfc9378

[16] M. Pattaranantakul, Q. Song, Y. Tian, L. Wang, Z. Zhang, A. Meddahi, and C. Vorakulpipat, “On Achieving Trustworthy Service Function Chaining,” *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 3140–3153, Sep. 2021.

[17] L.-J. Pang and Y.-M. Wang, “A New (t, n) Multi-Secret Sharing Scheme Based on Shamir’s Secret Sharing,” *Applied Mathematics and Computation*, vol. 167, no. 2, pp. 840–848, Aug. 2005.

[18] H. Sharaf, I. Ahmad, and T. Dimitriou, “Extended Berkeley Packet Filter: An Application Perspective,” *IEEE Access*, vol. 10, pp. 126 370–126 393, 2022.

[19] A. Mayer, S. Salsano, P. L. Ventre, A. Abdelsalam, L. Chiaraviglio, and C. Filsfils, “An Efficient Linux Kernel Implementation of Service Function Chaining for Legacy VNFs Based on IPv6 Segment Routing,” in *Proc .of IEEE Conference on Network Softwarization (NetSoft)*, pp. 333–341.

[20] M. Haeberle, B. Steinert, M. Weiss, and M. Menth, “A Caching SFC Proxy Based on eBPF,” in *Proc. of International Conference on Network Softwarization (NetSoft)*, Jun. 2022, pp. 171–179.

[21] A. Aguado, D. R. Lopez, V. Lopez, F. de la Iglesia, A. Pastor, M. Peev, W. Amaya, F. Martin, C. Abellan, and V. Martin, “Quantum Technologies in Support for 5G Services: Ordered Proof-of-Transit,” in *Proc. of European Conference on Optical Communication (ECOC 2019)*, Sep. 2019, pp. 1–3.

[22] E. S. Borges, V. B. Bonella, A. J. D. Santos, G. T. Menegueti, C. K. Dominicini, and M. Martinello, “In-situ Proof-of-Transit for Path-Aware Programmable Networks,” in *Prof. of IEEE International Conference on Network Softwarization (NetSoft)*, pp. 170–177.

[23] E. S. Borges, M. Martinello, V. B. Bonella, A. J. dos Santos, R. L. Gomes, C. K. Dominicini, R. S. Guimarães, G. T. Menegueti, M. Barcellos, and M. Ruffini, “PoT-PolKA: Let the Edge Control the Proof-of-Transit in Path-Aware Networks,” *IEEE Transactions on Network and Service Management*, pp. 1–11, 2024.

[24] FD.io, “FD.io VPP: VPP Inband OAM (iOAM),” https://docs.fd.io/vpp/19.01/ioam_plugin_doc.html, 2024, Accessed 18 Nov. 2024.

[25] C. Dominicini, D. Mafioletti, A. C. Locateli, R. Villaca, M. Martinello, M. Ribeiro, and A. Gorodnik, “PolKA: Polynomial Key-based Architecture for Source Routing in Network Fabrics,” in *Proc. of IEEE Conference on Network Softwarization (NetSoft)*, Jun. 2020, pp. 326–334.

[26] K. Bu, A. Laird, Y. Yang, L. Cheng, J. Luo, Y. Li, and K. Ren, “Unveiling the Mystery of Internet Packet Forwarding: A Survey of Network Path Validation,” *ACM Computing Surveys*, vol. 53, no. 5, pp. 104:1–104:34, 2020.

[27] D. Soldani, P. Nahi, H. Bour, S. Jafarizadeh, M. F. Soliman, L. Di Giovanna, F. Monaco, G. Ognibene, and F. Risso, “eBPF: A New Approach to Cloud-Native Observability, Networking and Security for Current (5G) and Future Mobile Networks (6G and Beyond),” *IEEE Access*, vol. 11, pp. 57 174–57 202, 2023.

[28] Y. Choe, J.-S. Shin, S. Lee, and J. Kim, “eBPF/XDP Based Network Traffic Visualization and DoS Mitigation for Intelligent Service Protection,” in *Proc. of Advances in Internet, Data and Web Technologies*, 2020, pp. 458–468.

[29] M. Xhonneux, F. Duchene, and O. Bonaventure, “Leveraging eBPF for Programmable Network Functions with IPv6 Segment Routing,” in *Proc. of the International Conference on emerging Networking EXperiments and Technologies*, Heraklion Greece, Dec. 2018, pp. 67–72.

[30] N. Van Tu, J.-H. Yoo, and J. Won-Ki Hong, “Accelerating Virtual Network Functions With Fast-Slow Path Architecture Using eXpress Data Path,” *IEEE Transactions on Network and Service Management*, vol. 17, no. 3, pp. 1474–1486, Sep. 2020.

[31] T. Hara and M. Sasabe, “Practicality of In-Kernel/User-space Packet Processing Empowered by Lightweight Neural Network and Decision Tree,” *Computer Networks*, vol. 240, p. 110188, Feb. 2024.

[32] C. Filsfils, D. Dukes, S. Previdi, J. Leddy, S. Matsushima, and D. Voyer, “IPv6 Segment Routing Header (SRH),” RFC 8754, Mar. 2020. [Online]. Available: https://www.rfc-editor.org/info/rfc8754

[33] KVM project, “Kvm,” https://linux-kvm.org/page/Main_Page, 2023, Accessed 18 Nov. 2024.

[34] libvirt, “Network XML format,” https://libvirt.org/formatnetwork.html#quality-of-service, 2023, Accessed 18 Nov. 2024.

[35] W. Tu, J. Stringer, Y. Sun, and Y.-H. Wei, “Bringing the Power of eBPF to Open vSwitch,” in *Proc. of Linux Plumbers Conference*, pp. 1–11.

Takanori Hara received the B.Eng. degree from National Institution for Academic Degrees and Quality Enhancement of Higher Education in 2016 and the M.Eng. and Ph.D. degrees from Nara Institute of Science and Technology, Japan, in 2018 and 2021. He is currently an Associate Professor with the Division of Information Science, Graduate School of Science and Technology, Nara Institute of Science and Technology, Japan. His research interests include eBPF/XDP, NFV, SDN, and AI for networking. Dr. Hara is a member of IEEE, ACM, and IEICE.

Masahiro Sasabe (Member, IEEE) received the B.S., M.E., and Ph.D. degrees from Osaka University, Japan, in 2001, 2003, and 2006, respectively. He is currently a Professor of Faculty of Informatics, Kansai University, Japan. His research interests include P2P/NFV networking, game-theoretic approaches, human-harmonized network systems, and network optimization. Dr. Sasabe is a member of ACM and IEICE.