ChainPatrol: Balancing Attack Detection and Classification with Performance Overhead for Service Function Chains Using Virtual Trailers

Momen Oqaily and Hinddeep Purohit, *CIISE, Concordia University;* Yosr Jarraya, *Ericsson Security Research;* Lingyu Wang, *CIISE, Concordia University;* Boubakr Nour and Makan Pourzandi, *Ericsson Security Research;* Mourad Debbabi, *CIISE, Concordia University*

https://www.usenix.org/conference/usenixsecurity24/presentation/oqaily

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

ChainPatrol: Balancing Attack Detection and Classification with Performance Overhead for Service Function Chains Using Virtual Trailers

Momen Oqaily[*] ✉, *Hinddeep Purohit*[*], *Yosr Jarraya*[†], *Lingyu Wang*[*] ✉, *Boubakr Nour*[†], *Makan Pourzandi*[†], *Mourad Debbabi*[*]
[*] *CIISE, Concordia University, Montreal, Canada, {momen.oqaily, hinddeep.purohit, lingyu.wang, mourad.debbabi}@concordia.ca*
[†] *Ericsson Security Research, Montreal, Canada, {yosr.jarraya, boubakr.nour, makan.pourzandi}@ericsson.com,*

Abstract

Network functions virtualization enables tenants to outsource their service function chains (SFCs) to third-party clouds for better agility and cost-effectiveness. However, outsourcing may limit tenants' ability to directly inspect cloud-level deployments to detect attacks on SFC forwarding paths, such as network function bypass or traffic injection. Existing solutions requiring direct cloud access are unsuitable for outsourcing, and adding a cryptographic trailer to every packet may incur significant performance overhead over large flows. In this paper, we propose *ChainPatrol*, a lightweight solution for tenants to continuously detect and classify cloud-level attacks on SFCs. Our main idea is to "virtualize" cryptographic trailers by encoding them as side-channel watermarks, such that they can be transmitted without adding extra bits to packets. We tackle several key challenges like encoding virtual trailers within the limited side channel capacity, minimizing packet delay, and tolerating unexpected network jitters. We implement our solution on Amazon EC2, and our experiments with real-life data and applications demonstrate that *ChainPatrol* can achieve a better balance between security (e.g., 100% detection accuracy and 70% classification accuracy) and overhead (e.g., almost zero increased traffic and negligible end-to-end delay) than existing works (e.g., up to 45% overhead reduction compared to a state-of-the-art solution).

1 Introduction

By decoupling network functions from proprietary physical boxes, Network Functions Virtualization (NFV) [1] allows tenants to host their network services on top of existing clouds managed by third-party cloud providers [2–4]. For instance, Amazon AWS Cloud is reportedly used to deploy an entire cloud-native 5G network [2] and VMware Telco Cloud platform is also designed for similar purposes [4]. However, outsourcing network services to third-party clouds may also bring novel security challenges. Since tenants typically have limited visibility into the underlying cloud infrastructure, they cannot directly inspect the cloud-level deployment of Service Function Chains (SFCs) to ensure their deployment matches the specification. Therefore, cloud-level integrity breaches may silently arise and stay invisible to tenants [5]. For instance, an attacker can exploit a vulnerability or a misconfiguration in cloud-level resources (e.g., virtual switches) either to attack the SFC forwarding path (e.g., skipping a firewall inside the SFC [6]), or to attack the traffic (e.g., packet/flow injection, dropping, and reordering [7]).

Many existing solutions for verifying forwarding paths (e.g., ICING [8], OPT [9], and EPIC [10]) are not directly applicable to SFCs as they are incompatible with the inherent characteristics of SFCs (i.e., paths dynamicity and packets mutability) [6,11]). Later works [6,7,12] address this through adding a cryptographic trailer to every packet, which can guarantee the integrity and detect various attacks. However, such capabilities come at a cost, i.e., the added communication overhead is usually proportional to the flow size (e.g., three times increase of the packet size [9] and 1.69 times increase of the network traffic size [10]). Such an overhead may be prohibitive for applications with large flow sizes, e.g., music streaming, video conferencing, and virtual reality, which have become increasingly popular today.

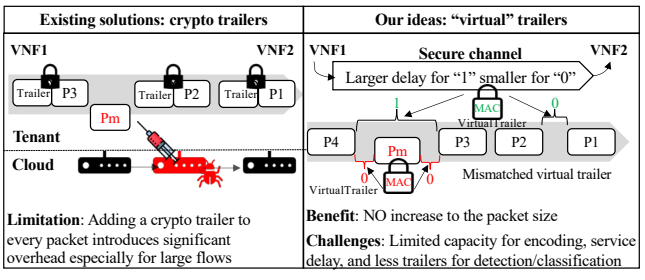

Figure 1: Example of crypto/virtual trailers

Example 1. Figure 1 shows an example to further illustrate the limitation of existing solutions (left), and our ideas (right). For simplicity, we consider a toy example of SFC con-

✉ Corresponding authors

sisting of two Virtual Network Functions (VNFs) connected through three cloud-level virtual switches, among which the middle one is compromised.

The Research Problem: The left side of Figure 1 illustrates a packet injection attack against the SFC. By exploiting either a vulnerability [13, 14] or misconfiguration [15] in the middle virtual switch, an attacker can inject a crafted packet (*Pm*) into the flow of normal packets (*P1*, *P2*, and *P3*) between *VNF1* and *VNF2*. Since the tenant has no direct access to cloud-level resources, it cannot easily uncover this attack. The protection provided by the cloud provider may also be limited since it is not necessarily aware of all tenants' SFCs and forwarding policies. Therefore, such attacks may fall into the gap and go undetected.

Existing solutions: The left side of Figure 1 also shows how existing cryptographic solutions (e.g., [6, 7, 12]) can detect the aforementioned attacks. Specifically, such solutions would modify *VNF1* to append a verifiable cryptographic trailer (including a Message Authentication Code (MAC) value computed over the packet and other trailer fields) to every packet before it leaves *VNF1*. *VNF2* is also modified to verify the trailer when the packet arrives. The malicious packet *Pm* can be reliably detected by *VNF2* since adversaries cannot forge such a trailer. However, as those solutions add a trailer to every single packet, they imply an overhead that is proportional to the flow size.

Our ideas: As shown on the right side of Figure 1, we "virtualize" the physical trailers as side channel watermarks, e.g., encoding a "virtual" trailer of '10' by slightly delaying packets *P1* and *P4* before they leave *VNF1*, such that the inter-packet delay between *P1* and *P2* becomes slightly smaller than usual, representing a '0' bit, while the delay between *P3* and *P4* becomes slightly larger than usual, representing a '1' bit. The injection of a malicious packet *Pm* can be detected as it will partially destroy our virtual trailer (there will now be two '0' bits after the injection, as shown in the figure). Moreover, as adversaries cannot forge the MAC (whose encoding is not shown in the figure and will be detailed in Section 3.2), they could not evade the detection with a fake trailer mimicking the expected inter-packet delays. Finally, the way the trailer is destroyed would also provide us additional information to not only detect the attack, but also to classify its type (detailed in Section 4).

Specifically, this paper presents *ChainPatrol*, a solution that applies the virtual trailer concept to SFCs for balancing security (i.e., attack detection and classification) with performance overhead (i.e., increased traffic and end-to-end delay). First, *ChainPatrol* offers a much lighter-weight solution than existing works based on physical trailers [6, 7, 12], since no extra bit needs to be added to packets (virtual trailers are encoded as side channel watermarks). Second, since virtual trailers contain similar information as in their physical counterparts, *ChainPatrol* can still guarantee the integrity of SFCs when the trailers are intact, and provide useful information for detecting and classifying attacks when the trailers are compromised (note these cannot be achieved by directly applying existing watermarking techniques [16–18], because the watermarks lack the semantics of a cryptographic trailer, e.g., sequence numbers of packets or flows and MAC values). Finally, unlike many existing works (which either require direct access to the underlying cloud infrastructure [19] or require modifications to the VNFs [20]), *ChainPatrol* provides a tenant-level solution that can be transparently integrated with existing SFCs, since IPDs are directly observable and mutable at tenant level and outside the VNFs. In summary, our main contributions are as follows.

- We propose the novel concept of *virtual trailer*, which inherits the advantages of both cryptographic trailer-based solutions (i.e., verifiable attack detection) and side-channel watermarking (i.e., lightweight). To realize this, we address several key challenges such as encoding virtual trailers within the limited capacity of a side channel, minimizing packet delay while computing virtual trailers, and handling unexpected network jitters. We believe this concept may potentially find other applications in a broader context.
- We apply the concept of the virtual trailer to design a tenant-based solution, *ChainPatrol*, for lightweight attack detection and classification in SFCs hosted on third-party clouds. First, *ChainPatrol* performs fast attack detection by identifying destroyed virtual trailers. Second, it performs in-depth attack classification through partial reconstruction of destroyed virtual trailers to match the expected ones. Finally, it verifies classification results through sharing a limited amount of information between the source and destination.
- We implement and deploy *ChainPatrol* based on Amazon EC2 and perform extensive experiments using both public datasets and an in-house 5G testbed. Our experimental results demonstrate the effectiveness (e.g., 100% detection accuracy and 70% classification accuracy) and efficiency (e.g., close to zero increased traffic and negligible end-to-end delay) of *ChainPatrol*. Our comparison of *ChainPatrol* to an existing physical trailer-based approach under real-world applications shows a significant reduction of communication overhead (up to 45%).

2 Background

This section provides background and our threat model.

2.1 SFC Forwarding Path Verification

Existing cryptographic solutions for forwarding path verification [6, 7] adapt traditional forwarding path verifica-

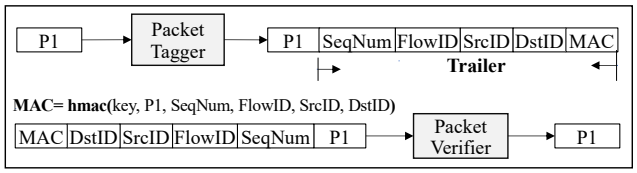

Figure 2: Example of a packet with cryptographic trailer [6]

tion protocols [8–10] to the NFV setting in order to meet its unique characteristics: (i) path dynamicity and unpredictability, (ii) packet modification by VNFs, and (iii) SFC migration and autoscaling. Specifically, those solutions intercept each egress packet at a source VNF to append a custom trailer to the packet, which consists of additional data fields along with a MAC computed over the entire packet content (including the trailer) using a secret key. A valid MAC extracted at the destination VNF (under the same key) would attest to the integrity of the packet content. This allows the traffic between VNFs to be verified for correctness.

Example 2. As shown in Figure 2, a packet tagger at the source VNF appends a packet trailer to each passing packet (top), and a packet verifier at the destination VNF verifies the packet and its trailer for integrity (bottom). First, `SeqNum` is a sequential number representing the ordering of packets between each pair of VNFs for a given flow. Together with `FlowID`, it is used to detect packet reordering and replay attacks within a given flow. Second, `FlowID` is uniquely mapped to the classic 5-tuple ⟨source IP address, destination IP address, source port, destination port, protocol⟩, and is used to detect flow-based attacks (e.g., flow injection and flow dropping). Third, `SrcID` and `DstID` refer to the source and destination VNFs, respectively, and are used to ensure compliance with the forwarding policy between the two VNFs. Finally, `MAC` is computed over both the packet content and above trailer fields and is used to authenticate the integrity of the packet and its trailer by the destination VNF.

2.2 Blind Watermarking

In this work, we adopt a *blind* watermarking approach (i.e., the packet verifier does not require knowledge about the original IPDs or watermarks). This choice was made for the following two reasons: (i) As we leverage watermarks to transmit virtual trailers without actually sending any bit, a non-blind approach, which must send information about the original IPDs and watermarks between the two VNFs, would defy our purpose. (ii) Although a non-blind watermarking approach is usually more resistant to network jitters (since the packet verifier knows the original IPDs), a blind approach is sufficient for our purpose, because the network connection between two VNFs in a data center is typically more stable (in contrast to the Internet for which most existing watermarking approaches are designed) [21].

Example 3. Figure 3 shows how a two-bit watermark message $\langle 1,0\rangle$ is encoded into the IPDs between four packets

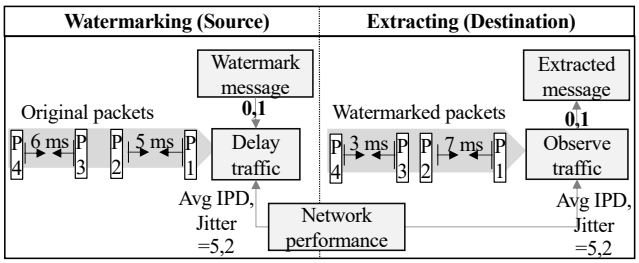

Figure 3: Example of watermarking and extracting two bits

at the source (left), and decoded at the destination (right). First, the average IPD (5 ms) and network jitter (2 ms) are continuously measured and used for reference on both sides. Second, the source encodes a bit 1 by increasing the original IPD between packets P_1 and P_2 corresponding to the summation of the average IPD and jitter (i.e., from 5 ms to 7 ms). Similarly, it encodes a bit 0 by decreasing the IPD between P_3 and P_4 corresponding to the difference between the average IPD and jitter (i.e., from 6 ms to 3 ms). Third, by comparing the observed IPD with the average IPD and jitter, the destination decodes a bit 1 between P_1 and P_2 as $7 \geqslant (5+2)$, and a bit 0 between P_3 and P_4 as $3 \leqslant (5-2)$. Note a jitter no greater than 2 would not change this result.

2.3 Threat Model

We consider a similar threat model as in recent works on crypto-based forwarding path verification [6, 7]. We also make assumptions related to our watermarking scheme.

In-Scope threats. Our in-scope threats include integrity breaches of the SFC forwarding paths or traffic caused by either (i) a malicious attacker compromising an underlying cloud-level forwarding device [14], or (ii) misconfigurations (intentionally or mistakenly) introduced by a cloud provider [15]. Specifically, we assume the network links between VNFs and the virtual switches used to steer traffic between such VNFs are both subject to the following attacks. First, a compromised cloud-level device may be used to skip VNFs or append malicious VNFs to the forwarding path, or to cause other unexpected forwarding decisions such as redirecting traffic intended for one VNF to another [7]. Second, a compromised cloud-level device may also be used to disrupt SFC traffic, such as injecting fake packets and dropping, modifying, reordering, or replaying packets. We assume the adversary may attempt to evade the detection through the so-called coward attack [9] (i.e., attacking selected flows not subject to detection) or attacks on the watermarking scheme used to encode virtual trailers (by deliberately altering the IPDs). A more detailed list of attacks covered in our work is given later in Table 2.

Out-of-Scope threats. By taking a tenant-based viewpoint, we assume no direct access to cloud-level resources or data, and therefore we must trust all the components to which the tenant has direct access. This includes the VNFs and

the gateway to access the SFC, and we also consider the cloud provider to be cheap-and-lazy but not malicious [15]. Attacks that can compromise those components or our solution itself (including the secret key used to compute the MAC) are out of scope of our work (which can be addressed through hardware-based solutions [6,12]). We focus on verifying the integrity of SFCs, and thus denial of service attacks (e.g., dropping all packets) and attacks on confidentiality or privacy of the traffic are out of scope. Moreover, since we leverage a fragile watermarking scheme, which relies on modified watermarks to detect attacks [22], watermark invisibility attacks [16]) are out of scope for our work. Finally, as demonstrated in Section 6, our solution is more beneficial for flows of relatively large sizes (for small flows, the overhead of physical trailers may be acceptable).

3 Virtual Trailer

This section defines the *virtual trailer* concept and details how to encode and decode virtual trailers based on IPDs.

3.1 Definitions

Our goal is to design virtual trailers to contain similar information as in their physical counterparts, such that they can support attack detection and classification (detailed in Section 4). However, there are several challenges. First, unlike a physical trailer which can be defined for (and added to) each packet, a virtual trailer can only be encoded in the IPDs of multiple packets. Therefore, we define a virtual trailer for each equal-sized group of consecutive packets within the same network flow, namely, a *block*. Second, due to the limited capacity of this side channel, we need to simplify the design of virtual trailers to only retain a minimal number of trailer fields, i.e., the identifiers of blocks and flows, while implicitly representing the source and destination VNFs inside the flow identifiers. Specifically, each flow identifier is now uniquely associated with a 7-tuple ⟨`source IP`, `destination IP`, `source port`, `destination port`, `protocol`, `source VNF`, `destination VNF` ⟩. Third, since a MAC value typically contains a much larger number of bits than what can be encoded inside a single block, we redefine the MAC to be computed over an equal-sized group of consecutive blocks inside the same flow, namely, a *SuperBlock*. Finally, to enable efficient attack classification and verification, the MAC is computed based on a Merkle hash tree [23] defined over the SuperBlock. The following first formally defines a virtual trailer for each block.

Definition 3.1 (Virtual Trailer) *Given block B_i ($1 \leqslant i \leqslant S_F$) inside a SuperBlock SB (with totally S_F blocks) in a flow F, the virtual trailer of B_i is a 3-tuple $VT_i = \langle BlockNum_i, FlowID, MAC_i \rangle$. First, $BlockNum_i \in \mathbf{N}$ is an integer value uniquely and sequentially assigned to each block in the given flow F. Second, $FlowID \in \mathbf{N}$ is an integer value uniquely and sequentially assigned to each flow. Third, $MAC_i \in \mathbf{N}$ is the $(\frac{i}{S_F})^{th}$ fraction of the Merkle tree [23] HMAC value computed over all the blocks in SB concatenated with their corresponding BlockNum and FlowID fields, i.e., $B_i||BlockNum_i||FlowID$ ($1 \leqslant i \leqslant S_F$).*

Table 1: Examples of Virtual Trailers (VT)

Flow	SB	B	VT		
			BlockNum	FlowID	MAC
<IP1,IP2,8,80,6,VNF1,VNF2>	1	1	0001	0001	0100
		2	0010	0001	1100
	2	3	0011	0001	1100
		4	0100	0001	1100
<IP1,IP2,4,80,6,VNF1,VNF2>	1	1	0001	0010	1010
		2	0010	0010	0101

Example 4. Table 1 shows an example with two flows, their SuperBlocks and blocks, and the corresponding virtual trailers (last three columns). The `BlockNum` is sequentially assigned to consecutive blocks inside the same flow, starting from a random number. Similarly, the `FlowID` is sequentially assigned to consecutive flows. For the first flow, the Merkle tree HMAC value of 01001100 is divided into two equal-sized bit strings (0100 and 1100) each of which forms the last field of the virtual trailer (similarly for the second flow). Figure 4 also depicts a virtual trailer (bottom right) and how the MAC value is computed (left) (the rest of the figure will be explained later in Example 5). Algorithm 1 in the appendix details the generation of virtual trailers.

3.2 Virtual Trailer Encoding

ChainPatrol intercepts packets at the egress of source VNF and determines their corresponding flows, SuperBlocks, and blocks (see Section 3.1). It then generates a virtual trailer for each block using Algorithm 1, and encodes the virtual trailer by modifying the IPDs (i.e., delaying one packet between every pair of packets, as explained in Section 2.2). Specifically, as each virtual trailer contains three fields (i.e., `BlockNum`, `FlowID`, and `MAC`), each block is divided into a series of three *frames*, and the IPDs in each frame are modified to encode a corresponding trailer field.

However, a key challenge lies in the encoding of the MAC trailer field. Specifically, since the MAC field of a virtual trailer is defined over a SuperBlock (as illustrated on the left side of Figure 4), it cannot be computed before the entire SuperBlock is received. Therefore, any received blocks of this SuperBlock must be delayed, since we do not yet know how to modify their IPDs. Such delays must be maintained until the last block arrives, after which we can then compute the virtual trailer, encode it by modifying the IPDs of all the buffered blocks, and finally forward all the blocks to the destination VNF. However, doing so would certainly cause prohibitive delay (proportional to the size of the SuperBlock). To address this, our key idea is to shift the encoding of each virtual trailer to the next SuperBlock. This allows us to compute the virtual trailer (which tells us how

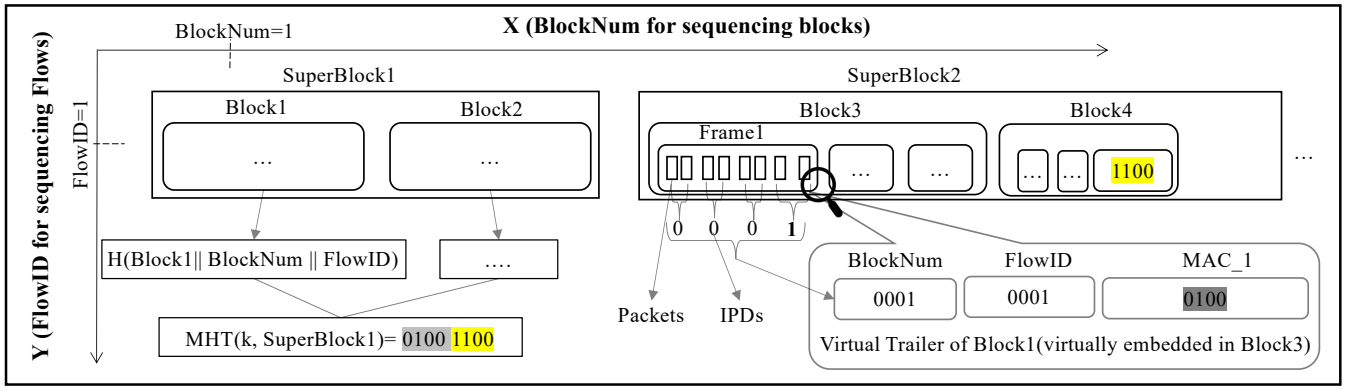

Figure 4: Example of a virtual trailer and its encoding

to modify the IPDs) ahead of the arrival of any block of the next SuperBlock, so we will know how to modify the IPD as soon as each such block arrives, and can forward the block with minimum delay (evaluated in Section 6).

Example 5. Figure 4 shows how the virtual trailer of a block (`Block1`) in the first SuperBlock (shown on the left) is encoded using a block (`Block3`) in the next SuperBlock (shown on the right). First, `Block3` is divided into three frames, each of which corresponds to a virtual trailer field. Second, the IPD between each pair of packets inside a frame is then modified (by delaying one of those packets) to encode each bit of the field. Similarly, the virtual trailer of `Block2` (i.e, the second block of `SuperBlock1`) is encoded in `Block4`. Note that, by the time `Block3` arrives, this virtual trailer of `Block1` would have already been computed. Therefore, we know how to modify the IPDs in `Block3` as soon as each pair of packets arrives, which can then be immediately forwarded.

Another challenge is to ensure the correct decoding of virtual trailers despite potential network jitters. For this purpose, *ChainPatrol* leverages an existing watermarking scheme [16] which is known to enable reliable extraction of watermarks at the destination side despite network jitters. The main idea is to introduce additional delay that is proportional to the expected level of jitters in order to cancel their impact on the encoded watermarks. Specifically, let $m_1 \cdots m_w$ be the w-bit representation of a virtual trailer field to be encoded. Denote the IPD between two packets arriving at time t_i and t_{i+1} as $IPD_i = (t_{i+1} - t_i)$. To encode bit m_i $(1 \leqslant i \leqslant w)$ using IPD_i, the new IPD, denoted by $nIPD_i$, is computed as: $nIPD_i = IPD_{AVG} + a \times M_i^e$, where IPD_{AVG} is the average IPD, a is the *watermarking amplitude* (computed using the Signal-to-Noise formula [16]), and $M_i^e = 1$ if $m_i = 1$, or $M_i^e = -1$ if $m_i = 0$ (whose effectiveness is further evaluated in Section 6). Algorithm 2 in the appendix details the encoding of the virtual trailers of a given SuperBlock.

3.3 Virtual Trailer Decoding

ChainPatrol passively monitors packets at the destination VNF and determines their corresponding flows, SuperBlocks, blocks, and frames. It then decodes the virtual trailers following a reversed process based on the observed IPDs. More specifically, let $rIPD_i$ $(1 \leqslant i \leqslant w)$ be the observed IPD between the i^{th} and $(i+1)^{th}$ packets, where w is the size of a virtual trailer field, and let IPD_{AVG} and a be the average IPD and the watermarking amplitude, respectively. Denote $M_i^d = (rIPD_i - IPD_{AVG})/a$. The w-bit binary representation of the virtual trailer field can be computed as: $m_i = 1$ if $M_i^d = 1$, or $m_i = 0$ if $M_i^d = -1$.

Attacks such as dropping, injection, or reordering of packets may shift the frames and blocks among the observed packets. Section 4 will detail how we address this issue and leverage it to classify attacks. Another challenge is related to the last SuperBlock. First, the virtual trailers of this last SuperBlock itself cannot be encoded using the IPDs of next SuperBlock (which does not exist). Second, when we divide a given flow into equal-sized blocks and SuperBlocks, the last SuperBlock may be incomplete, i.e., there are not enough packets remaining to compose a complete SuperBlock, which would prevent encoding the virtual trailers of the previous SuperBlock using IPDs. *ChainPatrol* addresses the first challenge by directly appending a physical version of the virtual trailers to the flow. Since *ChainPatrol* is designed for large flows (as stated in Section 2.3) with a significant number of SuperBlocks, the overhead of one trailer will be negligible in contrast to the flow size. To address the second challenge, *ChainPatrol* performs one of the following two options that introduce less overhead, i.e., (i) adding dummy packets to the last SuperBlock such that it can have enough IPDs for encoding the virtual trailers of the previous SuperBlock, or (ii) appending the physical trailers. Algorithm 3 in the appendix details virtual trailer decoding.

3.4 Handling Unexpected Network Jitters

ChainPatrol is designed for SFCs hosted inside cloud data centers, which is known to be more stable than the Internet [21], so network disturbance is expected to be rare. Nonetheless, since unexpected network jitters can still happen and disrupt the transmission of virtual trailers, we de-

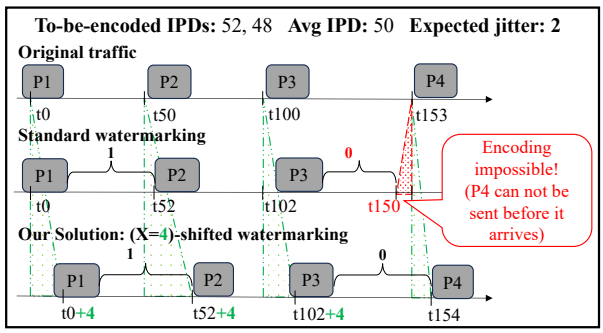

Figure 5: X-shift for encoding w/ unexpected IPD variation

sign *ChainPatrol* to provide three protection mechanisms against such jitters. First, as detailed in Section 3.2, our virtual trailer encoding scheme can already tolerate jitters through calculating and continuously updating the watermarking amplitude to be proportional to the average jitter. Second, we further design two mechanisms for the source and destination VNFs, respectively, to handle larger and unexpected jitters. Finally, we experimentally evaluate the effectiveness and overhead of those methods in Section 6.

The X-shift method. An unexpected amount of jitter exceeding the watermarking amplitude may render the encoding challenging for the source VNF. Specifically, if the second packet in a pair arrives so late, that it has not arrived when the VNF needs to send it to match the to-be-encoded IPD, then encoding becomes impossible since we cannot send a packet that has not yet arrived. For instance, Figure 5 shows two watermark bits 1,0 to be encoded with new IPDs 52,48 (assuming an average jitter of 2 ms) between two pairs of packets ($P1$ to $P4$). The top timeline depicts their actual arrival time, which shows an unexpected delay of 3 ms between $P3$ and $P4$. As the middle timeline shows, a standard encoding scheme will fail to encode the second bit, since $P4$ is supposed to depart at t=150 (to have an IPD of 48) but by that time it has not yet arrived. One naive solution here is for both VNFs to use two pre-defined high IPD values (which will never occur naturally) to encode 0 and 1, respectively, such that $P4$ can now be further delayed to encode 0. However, this could lead to significant delay considering that the watermarks in our application are generally large in size so this scenario may occur very often. To address this, we adopt a pragmatic X-shift approach as follows. As the lower axis in Figure 5 shows, the first packet in each pair will be proactively delayed by a small amount ($X = 44$ ms in this case) to minimize the likelihood of requiring the costly solution of using pre-defined large IPDs. The source-side VNF periodically updates X based on observed IPD variation (note, unlike watermarking amplitude, the destination VNF does not need to know X). Finally, the impact of delaying the first packet of each pair will not add up and hence remains negligible for the entire flow.

The α-amplitude method. An unexpected amount of jitter exceeding the watermarking amplitude may also prevent the correct decoding of virtual trailers at the destination VNF. Specifically, the jitter may cause the flipping of some bits in the virtual trailer during transmission, which can generate false positives for attack detection. To address this issue, we adopt the pragmatic approach of using a larger-than-necessary watermarking amplitude upon observing unexpected jitters. Specifically, the watermarking amplitude (as calculated in Section 3.2) will be multiplied with a parameter $\alpha(\alpha \geqslant 1)$, which is decided based on the observed level of jitter variation (i.e., observing a larger variation leads to a larger α being used). Using such a larger-than-necessary amplitude creates a bigger gap between the IPDs used to encode watermark bits 0 and 1, which ensures the destination VNF can still correctly distinguish between them and thus reduces the chance of a bit flipping caused by unexpected jitters. Finally, the watermarking amplitude, the X value, and the α parameter will all be constantly monitored and continuously updated to reflect changing network states (detailed in Section 5) and their effectiveness will be confirmed in Section 6.

4 Attack Detection and Classification

This section details how *ChainPatrol* detects and classifies various attacks using virtual trailers.

4.1 Attack Detection

Similar to existing works using physical trailers [6,7], *ChainPatrol* detects SFC attacks by matching decoded virtual trailers with corresponding blocks. However, the unique design of virtual trailers (detailed in Section 3) leads to two differences as follows. First, since the virtual trailers of one SuperBlocks are always encoded in the IPDs of next SuperBlock, *ChainPatrol* iteratively performs attack detection inside a moving window that slides over the next two consecutive SuperBlocks of the current flow in each iteration. Second, recall that the HMAC of a SuperBlock is computed based on a Merkle hash tree defined over all the blocks, and each virtual trailer only includes part of this HMAC value (as illustrated in Figure 4). Therefore, *ChainPatrol* first decodes the virtual trailers encoded in the second SuperBlock inside the moving window, and recomputes the Merkle tree HMAC based on the decoded `BlockNum` and `FlowID` values and all packets of the first SuperBlock. It then compares this re-computed HMAC with the decoded HMAC value (obtained by concatenating the `MAC` fields of all the virtual trailers decoded from the second SuperBlock).

One challenge is that, since an attack may cause the frames, blocks, or SuperBlocks to shift, *ChainPatrol* needs to identify the beginning of the next SuperBlock once an attack is detected. Specifically, if all the virtual trailer fields match the first SuperBlock in the window, *ChainPatrol* marks the

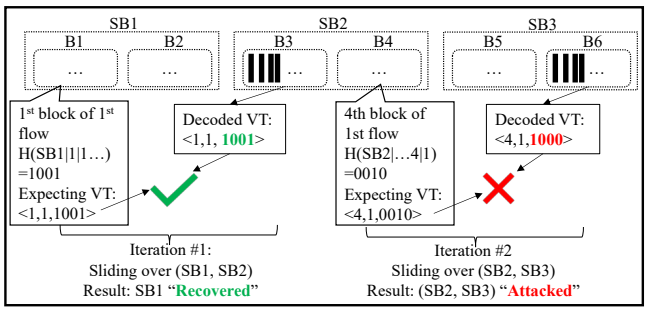

Figure 6: Example of attack detection

first SuperBlock as "recovered", and the window slides forward by one complete SuperBlock. When an attack is detected, *ChainPatrol* marks the current pair of SuperBlocks as "attacked" (which will be further inspected for attack classification), and the window slides forward by only a pair of packets in order to identify the beginning of the next SuperBlock. Once a new SuperBlock is identified, *ChainPatrol* resumes the normal attack detection as described above. Algorithm 4 in the appendix details the attack detection.

Example 6. Figure 6 (top) shows the first three SuperBlocks of the first flow, and (bottom) an example of attack detection.

- In the first iteration, the window slides over the first two SuperBlocks, `SB1` and `SB2`. For the first block of `SB1`, the expected `BlockNum` = 1) and `FlowID` = 1), as this is assumed to be the first block of the first flow. As the figure shows, the first virtual trailer decoded from `SB2` also contains the same `BlockNum` and `FlowID` values.
- The expected `MAC` needs to be computed over the content of *SB*1 concatenated with the decoded `BlockNum` and `FlowID` fields of both blocks inside `SB1`. As the figure shows, the first half of the MAC value decoded from `SB2` matches the expected value (1001). Assuming the second half also matches, `SB1` can be marked as "recovered" as no attack is detected.
- In second iteration, the window slides to `SB2` and `SB3`. Similarly, the expected `BlockNum` and `FlowID` match the ones decoded from `SB3`. However, assume the first half of the expected MAC value (0010) does not match the decoded one (1000), an attack is thus detected. At this point, it is unclear whether the attack happened to `SB2` or the virtual trailers (i.e., IPDs) in `SB3`, so both SuperBlocks are marked as "attacked" for classification.
- Since the attack may have caused packets to be injected, dropped, or reordered, we cannot take the beginning of the next SuperBlock for granted. Instead, the window can only slide forward by one pair of packets at a time, until we identify the next intact SuperBlock.

4.2 Attack Classification

Upon the detection of an attack, *ChainPatrol* classifies it with one of the known attack types listed in Table 2. In addition to the well-known packet-level attacks reported in the literature, *ChainPatrol* also considers block-level attacks that involve the manipulation of a whole packet block, flow-level attacks involving all packet blocks in a flow, and SFC-level attacks targeting the forwarding paths (instead of packets). To classify a SuperBlock marked as attacked during the detection stage (Section 4.1), *ChainPatrol* attempts to reconstruct the expected virtual trailers by applying each of the classification rules listed in the last column of Table 2, and it classifies with an attack type if the reconstruction is successful under the corresponding rule.

Table 2: Attack types covered by *ChainPatrol*

	Attack type	Classification rules based on virtual trailers
Packet	Injection	Unrecovered VT & larger block size
	Reordering	Partially recovered VT & expected block size
	Replay	Unrecovered VT & larger block size
	Modification	Unrecovered VT with mismatched MAC
	Dropping	Partially recovered VT & smaller block size
Block	Injection	Unrecovered VT & expected block size
	Reordering	Out of order `BlockNum`
	Replay	Recovered VT with repeated `BlockNum`
	Dropping	Missing VT with specific `BlockNum`
Flow	Injection	Unrecovered `FlowID`
	reordering	Out of order `FlowID`
	replay	Repeated VT for all blocks
	dropping	Missing `FlowID`
SFC	VNF Inj.	Unrecovered `FlowID`
	Reordering	Out-of-order `FlowID`
	VNF-Loop	Repeated `FlowID`
	Skipping	Missing `FlowID`

One challenge is that, depending on the level of the classification rule (first column of Table 2), this reconstruction may involve two SuperBlocks, the remainder of the flow, or even other flows (e.g., flow or SFC-level attacks). Therefore, the attack classification naturally requires more effort than attack detection. This explains why *ChainPatrol* adopts a layered approach to separate the (faster) detection from (slower) classification such that it can provide faster detection (as shown through experiments in Section 6).

Another challenge is that the missing information (e.g., in case of dropping or modification attacks) and lower granularity of virtual trailers (i.e., per block instead of per packet) together may prevent a full reconstruction of virtual trailers, and hence the attack classification result can no longer be guaranteed like with detection. To address this, we leverage our design choice of computing the MAC based on a Merkle hash tree [23] defined over a superblock (detailed in Section 3) to enable efficient source-assisted verification of the classification result in such cases. Specifically, the well-known property of a Merkle hash tree [23] allows us to verify the MAC (and hence the correctness of classification) by requesting selected tree nodes from the source, e.g., a single node (common ancestor) is sufficient for $log(N)$ consecutive blocks (in contrast, N nodes would be requested if the MAC were computed directly over the SuperBlock). Algorithm 5 in the appendix details the attack classification.

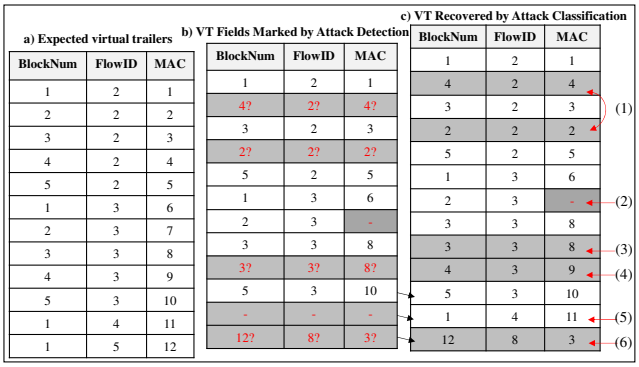

Figure 7: Examples of attack classification (a question mark (?) means mismatching with expected value; a dash (-) means no received packet; grey rows are involved in detection or classification; numbers (1-6) indicate attacks)

Example 7. Figure 7 depicts six examples of classification:

- Attack (#1) is classified as reordering of the two shaded blocks, since their `BlockNum` fields are out-of-order (Figure 7.c). Specifically, attack classification attempts to reconstruct the expected virtual trailers (i.e., first four rows in Figure 7.a) by applying each classification rule (last column of Table 2) to the attacked virtual trailers in Figure 7.b. The rule for block reordering leads to a successful reconstruction, and hence the attack is classified as such.
- Attack (#2) is classified as packet-dropping, since attack classification can partially reconstruct the virtual trailer for the seventh row (where the `BlockNum` and `FlowID` fields are intact) and beyond, while the block size is smaller than expected (the `MAC` field is missing), which matches the classification rule for packet dropping in Table 2. This result can be further verified by requesting the missing information (Merkle tree node).
- Attacks (#3) and (#4) are similarly classified as packet block replay and drop, respectively.
- Attack (#5) is a flow drop attack, as the expected `FlowID` is four (end of Figure 7.a) is missing in Figure 7.c, which matches the classification rule for flow dropping.
- Attack (#6) is a VNF skipping attack since a packet block received at the destination VNF (the last row in Figure 7.b) has a missing FlowID (which does not appear in Figure 7.a), which matches the classification rule for VNF skipping.

5 Implementation

ChainPatrol is composed of two *agents* per pair of communicating VNFs, plus a central *orchestrator*. The agents perform watermark encoding/decoding, attack detection, and local attack classification. The orchestrator is in charge of managing the agents and performing global attack classification. *ChainPatrol* is implemented in C++ programming language, with approximately 1300 lines of code at the agent, and 200 lines at the orchestrator. The following details the implementation and challenges.

***ChainPatrol* agent.** Each agent includes four components: 1. The *controller* performs the following functionalities. First, it receives *ChainPatrol* parameters from the orchestrator at initialization, and it then continuously monitors the network performance (*e.g.*, IPD and network jitter) to compute the watermark amplitude and X-shift values. Second, at the source VNF, it intercepts the egress traffic, generates virtual trailers (the Merkle hash trees are stored inside shared storage in case the destination-side agent may need selected nodes, as discussed in Section 4.2), triggers the watermarker, and forwards the traffic. Third, at the destination VNF, it passively monitors the ingress traffic to measure the IPDs, triggers the watermarker, attack detector, and attack classifier, and finally reports the detection and classification results to the orchestrator. 2. The source-side *watermarker* encodes the virtual trailers generated by the controller using Algorithm 2, and the destination-side watermarker decodes the virtual trailers from the ingress traffic using Algorithm 3. 3. Once triggered by the controller, the *attack detector* applies Algorithm 4 to mark SuperBlocks as either recovered or attacked. 4. Finally, once triggered, the *attack classifier* applies Algorithm 5 to classify the detected attacks using classification rules that only involve local traffic.

***ChainPatrol* orchestrator.** The orchestrator includes three components: (1) The *agents manager* is responsible for instantiating the agents and communicating with them. (2) The *attack classifier* is responsible for global attack classification that cannot be performed locally at each agent (e.g., flow or SFC-level attacks). (3) The *Audit Table* is used to store the detection and classification results both from the agents and from the orchestrator. At the initialization of *ChainPatrol*, the orchestrator shares with the agents the following *ChainPatrol* parameters: 1. SuperBlock size and block size per flow type (as different numbers of packets can be exchanged per flow type, for example, HTTP vs. SSH). 2. Initial seed values per pair of agents (used with a pseudo-random generator for virtual trailer generation). 3. A cryptographic key per pair of agents (for computing the MAC).

Avoiding VNF instrumentation. To simplify deployment, *ChainPatrol* intercepts egress traffic from the source VNF for encoding and monitors ingress packets at the destination VNF for decoding, without requiring instrumentation of the VNFs. This is achieved using raw sockets to attach agents as proxies to VNF interfaces, allowing traffic monitoring and forwarding. Additionally, iptables are used at the source to ensure only watermarked traffic forwarded by ChainPatrol reaches the destination VNF.

Minimizing communications. *ChainPatrol* minimizes

communication overhead by: (1) The agents at source and destination VNFs independently generate virtual trailer fields using common seed values obtained at initiation, avoiding ongoing communication. (2) Agents and orchestrator communicate via shared storage, leveraging Amazon ElastiCache for Memcached [24]. (3) AWS Lambda [25] facilitates access to Memcached, enhancing efficiency.

Root privileges and control packet filtering. While working with raw sockets, we faced two challenges. First, the agent couldn't attach to VNFs due to lacking root privileges, causing silent failures. To solve this, we split the code into privileged and non-privileged chunks, applying setuid for the former. Second, the destination-side agent struggled to decode virtual trailers due to confusion from control packets mixed with data traffic. We address this issue by filtering out control packets through inspecting the `PSH (Push) flag` field in the TCP header of packets (control packets have this flag set to `zero`, while data packets set to `one` for the packets to be delivered without buffering).

Multithreading. *ChainPatrol*'s layered design separates faster watermark encoding/decoding from slower attack detection. Implementing agents with a single thread would cause unacceptable delays, as the controller must wait for tasks to finish before forwarding packets. Thus, we use multiple threads: one for capturing packets, generating virtual trailers, and encoding them; and two more for independent attack detection and classification. The controller component ensures necessary thread execution serialization.

6 Performance Evaluation

This section presents the evaluation of *ChainPatrol*.

6.1 Evaluation Environment

Testbed setup. We deploy *ChainPatrol* in Amazon EC2 using EC2 instances of the `t3a.small` type (*i.e.*, 2 vCPUs of 2.50 GHz AMD EPYC 757, 2 GB RAM, up to 5Gbps of network bandwidth). Each instance runs Amazon Linux 2 (based on Ubuntu 22.04). We leverage the *Memcached database cluster*[1] (a high-performance, distributed memory object caching system) running on a `t3.micro` instance (2 vCPUs of 3.10 GHz Intel Xeon Scalable processor, Skylake 8175M or Cascade Lake 8259CL, 1 GB RAM, up to 5Gbps of network bandwidth) as shared storage between the *ChainPatrol* orchestrator and agents. We additionally use *Lambda Functions*[2], written in Python 3, as the interface between *ChainPatrol* and the Memcached cluster. Our evaluation focuses on data plane TCP flows between VNFs inside a SFC, although *ChainPatrol* can work for other types of traffic.

[1] Amazon ElastiCache: https://aws.amazon.com/elasticache/memcached/

[2] AWS Lambda: https://aws.amazon.com/lambda/

Datasets. We evaluate *ChainPatrol* based on: (i) a public dataset [26], (ii) our Free5GC/Kubernetes-based 5G testbed deployed on Amazon EC2, and (iii) the packet sizes of real-world applications in [27, 28] (for comparing to a state-of-the-art solution [6]). First, since IPD has the most significant impact on performance, we use the realistic IPD values from a public cyber defense dataset [26] for our first dataset (Dataset1). Second, to obtain live data from a real cloud, we deploy our own 5G core testbed on Amazon EC2 to generate a large live-streaming dataset with up to 600-750 packets per second for our second dataset (Dataset2). As shown in Table 3, those two datasets demonstrate distinct ranges of IPDs (80 ms vs. 7 ms) and jitters, which are representative of traditional networks (based on physical infrastructures) and virtual networks (based on containers and virtual machines), respectively. Finally, we use the packet sizes of several real-world applications in [27, 28] to compare with AuditBox [6].

Table 3: Datasets description

Dataset	IPD range	Source	Application	Rate
Dataset1	70-90ms	Public	Cyber defense	600K/s
Dataset2	5-10ms	Amazon EC2	5G core	500K/s

6.2 Experimental Results

We first evaluate the impact of various *ChainPatrol* parameters on its effectiveness and overhead. Then, we measure the accuracy and efficiency of its attack detection and classification. Finally, we compare *ChainPatrol* with a state-of-the-art physical trailer-based solution, namely, AuditBox [6].

6.2.1 Parameters Evaluation

We study how watermarking parameters may affect its effectiveness, and how *ChainPatrol* parameters may impact the service delay and its overhead.

Watermarking effectiveness. Figure 8.A and Figure 8.B show the impact of the two watermarking parameters, i.e., the watermarking amplitude (Section 3.2) and the X-shift value (Section 3.4), on the watermark extraction rate (the standard metric for watermarking effectiveness [29]). First, Figure 8.A shows that a watermark amplitude value of 1.2 (for Dataset1) and 1.6 (for Dataset2) can already achieve 100% extraction rate. The results also show that Dataset2 generally requires a slightly larger amplitude value than Dataset1 (this can be explained by the relatively higher percentage of jitter in Dataset2, as shown in Table 3). Second, Figure 8.B shows that a larger X-shift value is required for Dataset1 to achieve 100% extraction rate (which implies this parameter is more closely related to the average IPD). Specifically, Datatset1, whose average IPD is 80 ms, requires $X \approx 3$; Dataset2, whose average IPD is 8 ms, requires $X \approx 0.5$. The results also show that, without our X-shift solution (i.e., $X = 0$), the extraction rate would be significantly

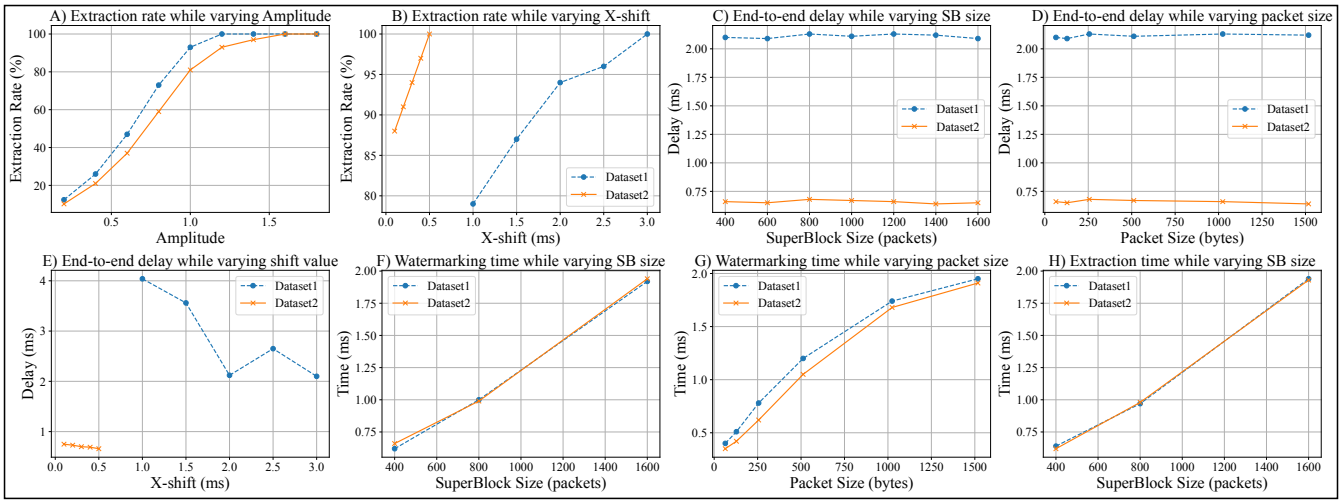

Figure 8: Parameters evaluation results

lower (less than 80% for Dataset1 and 90% for Dataset2). Note in both experiments, we fix the other parameter (at a value achieving 100% extraction rate).

Service delay. In this set of experiments, we study how the end-to-end service delay may be affected by different *ChainPatrol* parameters. Figure 8.C and Figure 8.D show the end-to-end service delay for different SuperBlock sizes (a parameter of *ChainPatrol*) and packet sizes (a characteristic of the traffic), respectively. Both results show that *ChainPatrol* introduces negligible end-to-end service delay on both datasets (around 2.1 ms for Dataset1 and 0.68 ms for Dataset2), and varying the SuperBlock and packet size has almost no impact on the end-to-end delay. This is mainly due to the facts that *ChainPatrol* agents mainly examine the headers (hence packet sizes have little impact), and the virtual trailers are always encoded in the next SuperBlock (see Section 3.2) so the time for generating virtual trailers (which depends on the SuperBlock size) does not affect packet processing (as the trailers are already ready when the packets arrive). Figure 8.E shows how the X-shift value affects the end-to-end delay. As we can see, a larger X value generally leads to a lower delay before it reaches the value that produces 100% extraction rate.

Overhead. In this set of experiments, we evaluate the overhead of *ChainPatrol* with respect to time. First, Figure 8.F and G show the average watermarking time (i.e., the time taken for a source-side agent to generate and encode virtual trailers) for different SuperBlock and packet sizes. The results show that the watermarking time increases almost linearly in SuperBlock size (as more packets need to be processed) and increases more slowly in packet size (as only the MAC field is affected), while the maximum watermarking time is less than $2ms$ for both datasets. Second, Figure 8.H shows the extraction time (i.e., the time taken for a destination-side agent to decode and verify virtual trailers) for different SuperBlock sizes. The results are very similar to Figure 8.F since the two agents perform similar but reversed operations (the results on extraction time vs. packet size are also very similar to Figure 8.G and omitted).

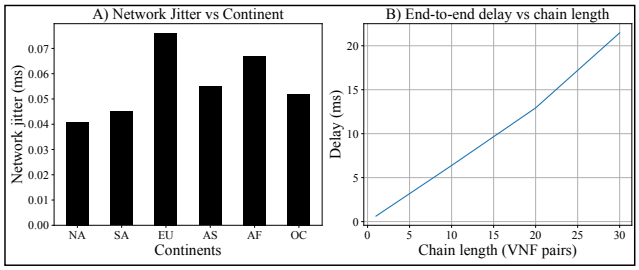

Figure 9: Parameters evaluation results (continued)

Jitter vs. AWS regions. In order to understand the potential impact of VNF locations on *ChainPatrol*, we measure the network jitter between our location (North America) and one region in each continent. We repeat the experiment over a period of several weeks and observe a relatively stable jitter over time. As shown in Figure 9.A, the jitter value within the same continent (North America) is the lowest. The highest value of jitter is observed between North America and Europe, which is roughly double the value of jitter within the same continent (in contrast, our experiments reported in Section 6.2.4 will use up to six times larger jitters). Finally, we also experiment with the Elastic Network Adapter (ENA) feature of AWS (which provides Enhanced Networking for EC2 instances), and our results show that jitters within the same continent drop from 0.175ms without ENA to 0.041ms with ENA. For the remaining experiments, we have used EC2 instances with ENA enabled.

End-to-End delay vs chain length. In this experiment, we evaluate the end-to-end delay introduced by *ChainPatrol* by varying the number of VNFs in the chain up to 30 VNF pairs (according to the literature [30], 25 VNF pairs represent a reasonably large NFV setup). As shown in Figure 9.B, the processing delay of *ChainPatrol* agents is roughly con-

stant for each pair (i.e., 0.63ms), and accumulative along the chain (which also holds for physical trailers). The maximum cumulative delay observed for a chain with 30 pairs of VNFs is about 22ms (which is barely noticeable for interactive music and not noticeable for games [31]). In practice, the length of a VNF chain is usually much smaller, and hence the end-to-end delay of *ChainPatrol* should be negligible.

Summary. It is relatively easy (i.e., with small amplitude and X values) for *ChainPatrol* to ensure the correctness of its decoding of virtual trailers due to the more stable nature of traffic between VNFs (as explained in Section 5, *ChainPatrol* also continuously adjusts those parameters based observed traffic). Moreover, our results show that *ChainPatrol* causes negligible end-to-end delay to services on both datasets, e.g., the entire flow will be delayed by only about 2.1 ms and 0.68 ms (2.6% and 9.8% of the normal delay between two packets), respectively. Finally, the overhead of *ChainPatrol* in terms of delay is negligible.

6.2.2 Attacks detection and classification

We evaluate the accuracy and efficiency of attack detection and classification.

Accuracy evaluation. First, Figure 10.A shows that *ChainPatrol* achieves 100% detection accuracy with up to 50 attacks of different types (as listed in Table 2) performed on both datasets. This is expected as the experiment is performed with the *ChainPatrol* parameters that can ensure 100% watermark extraction rate (detailed in Section 6.2.1), and the extracted virtual trailers possess similar cryptographic properties as physical trailers to guarantee accurate detection (detailed in Section 4.1). Second, Figure 10.B shows the attack classification accuracy. To evaluate the true capability of virtual trailers for attack classification, this experiment deliberately skips the source-assisted verification step (described in Section 4.2), such that all inaccurate classification results will be counted. The results show that the attack classification accuracy stays almost fixed at around 70% under different amounts of attacks for both datasets. In contrast to detection, classification shows a lower accuracy. This is expected due to missing information caused by attacks and potential collision between attack types (e.g., a packet reordering attack may impact virtual trailers same as a combination of packet dropping and injection attacks).

Efficiency evaluation. First, we evaluate the attack detection time while varying the block size and the number of compromised blocks, respectively. In Figure 10.C, the total number of compromised blocks is fixed at 30, and we measure the time required to detect those compromised blocks. As the results show, the detection time increases almost linearly in the block size, with less than 2.4 seconds required for all block sizes for both datasets (no significant difference between the datasets). Note the detection time has no impact on the service delay (due to our multithreading implementation, as detailed in Section 5), and the second-level detection time is reasonable since the detection results are meant to be inspected by human experts. Second, we evaluate the time required to detect different amounts of block-level attacks while fixing the total number of SuperBlocks at 50. In Figure 10.D, the detection time increases almost linearly in the number of attacks, with up to 5.5 seconds required for detecting all 50 attacks.

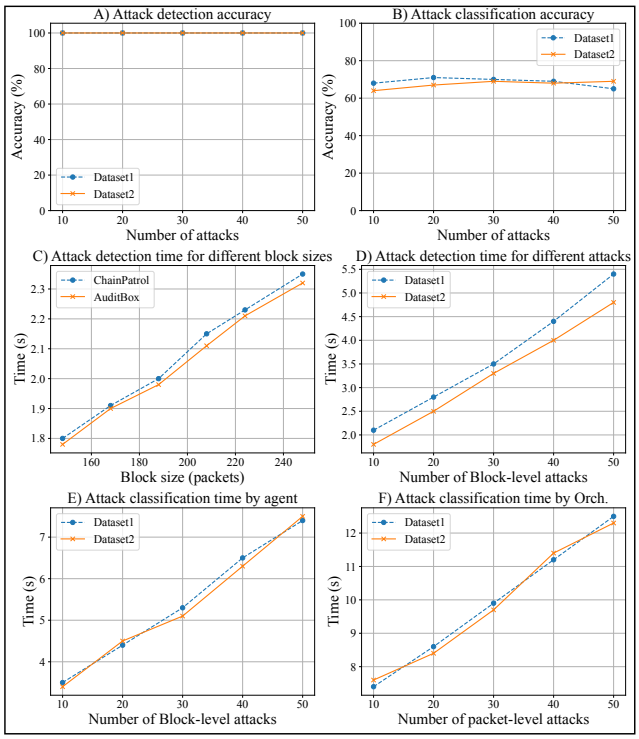

Figure 10: Attacks detection and classification results

Next, we evaluate the attack classification time by both the agents and the orchestrator, for block-level attacks and packet-level attacks, respectively, while varying the number of attacks. First, in Figure 10.E, the total number of SuperBlocks is fixed at 50, and we measure the time required to classify different amounts of block-level attacks locally by the destination-side agent. As the results show, the local classification time increases almost linearly in the number of block-level attacks, with around 7.5 seconds required for classifying all the 50 block-level attacks, which is slightly higher than the time for detection (i.e., 5.5 seconds). Second, Figure 10.F shows the global classification time taken by the orchestrator for classifying different amounts of packet-level attacks. The results show a similar linear trend, with around 12.5 seconds required for classifying all the 50 attacks. The orchestrator is taking more time since the global classification it performs involves more input information, and the packet-level attack classification rules may require the orchestrator to slide forward more slowly (by two packets, instead of a block, as detailed in Section 4.2).

Summary. Virtual trailers can guarantee 100% detection accuracy like their physical counterparts do. Virtual trailers provide a lower (70%) but still acceptable classification accuracy, since the result will be further verified in the source-assisted verification step described in Section 4.2. Our results also show attack detection and classification to be efficient (both take a few seconds for 50 attacks), scalable (both show a linear trend), and lightweight (both consume negligible resources).

6.2.3 Comparison with existing work

We compare *ChainPatrol* to a state-of-the-art physical trailer-based solution, namely, AuditBox [6] (which is re-implemented to ensure the two solutions can be compared under the same environments and settings). We compare them using Dataset2 (which is more representative of SFC applications) with packet sizes varied based on different real-world applications in two public datasets [27, 28]. Table 4 shows a general comparison in terms of end-to-end delay and communication overhead. Overall, AuditBox [6] has less delay and higher communication overhead than *ChainPatrol* on both datasets. Between the two datasets, the end-to-end delay of *ChainPatrol* stays the same while AuditBox incurs slightly more end-to-end delay for [27]. The communication overhead of *ChainPatrol* stays almost zero for both datasets while AuditBox incurs slightly more communication overhead for [28]. The different results of AuditBox between the two datasets can be explained by the different packet sizes (572 bytes on average for [27] and 85 bytes for [28]) due to different applications (e.g., audio/video in [27] v.s. Chat and VOIP in [28]).

More specifically, Figure 11.A shows the top four applications for which *ChainPatrol* achieves the most reduction in communication overhead (between 45% for DNS and 22% for Chat) compared to AuditBox. To further study the impact of packet sizes on overhead, we compare the overhead added by *ChainPatrol* and AuditBox while varying the packet size (with fixed flow size). Figure 11.B shows that *ChainPatrol* has almost zero overhead across all packet sizes (as *ChainPatrol* only adds a negligible amount of bits at the end of each flow, as discussed in Section 3.2), while the overhead of AuditBox's physical trailers ranges from around 10% (for large packet sizes up to 512 bytes) to 200% (for small packet size of 20 bytes). Next, Figure 11.C shows an average end-to-end delay between 0.35 and 0.55 ms for AuditBox, and 0.65 to 0.68 ms for *ChainPatrol*. The relatively higher delay of ChainPatrol is expected due to its more complex design of virtual trailers (e.g., a virtual trailer is computed over a group of packets, whereas a physical trailer is over a single packet). Nonetheless, the delay has a negligible impact on application performance (see Section 6.2.1).

Next, we compare *ChainPatrol* to an extension of Auditbox, namely, *density reduction trailer*, in which each physical trailer is computed over, and added to a group of packets

Table 4: General comparison in terms of delay and overhead

	Delay		Overhead	
	Dataset [27]	Dataset [28]	Dataset [27]	Dataset [28]
ChainPatrol	0.67 ms	0.67 ms	1.0 X	1.0 X
AuditBox [6]	0.48 ms	0.38 ms	1.12 X	1.38 X

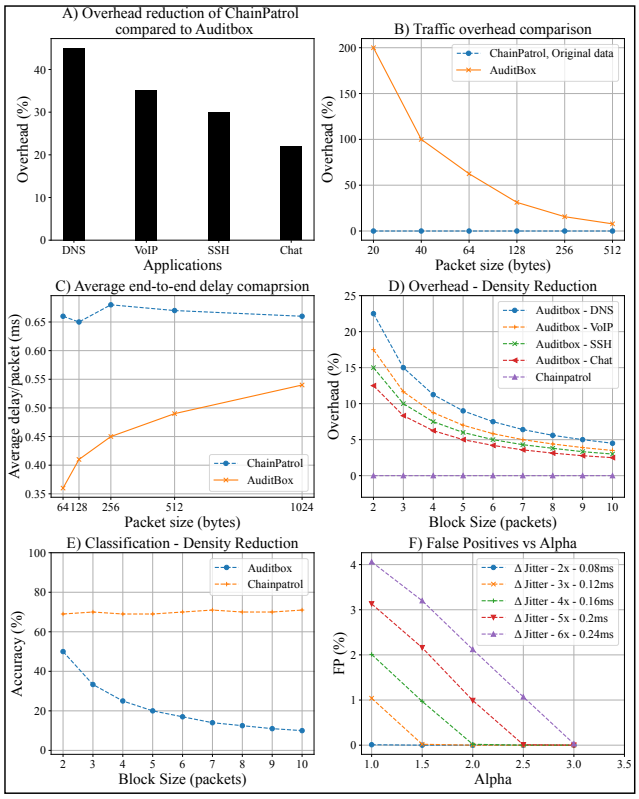

Figure 11: Detailed comparative evaluation results

(i.e., block) to reduce the overhead. As Figure 11.D shows, *ChainPatrol* overhead stays around zero regardless of the block size, while Auditbox constantly shows higher overhead even though density reduction trailers can reduce it. Specifically, as the block size increases from 2 to 10 (packets), the Auditbox overhead decreases from around 23% to 4.5% (45% without density reduction). On the other hand, as Figure 11.E shows, while *ChainPatrol* can maintain a stable classification accuracy when block size increases, the accuracy of Auditbox drops quickly. This can be explained by the fact that a MAC value stored physically can only indicate a "match"or "no match" (to the packet and trailer content), which helps attack detection but not classification. In contrast, a MAC value virtually stored (encoded) inside (the IPDs of) multiple packets can additionally provide useful information to help attack classification. For instance, if a packet is dropped, the mismatched MAC in a (density reduction) physical trailer cannot provide any information about which packet inside the block is dropped. In contrast, since this dropped packet is also used to encode part of the MAC value for the previous SuperBlock (detailed in Section 3.2),

comparing the expected MAC value (recomputed from the previous SuperBlock) to the current SuperBlock can easily pinpoint which packet is missing (similar for other attacks).

Summary. The comparison between AuditBox [6] and *ChainPatrol* shows that *ChainPatrol* introduces almost no extra communication overhead regardless of the packet or flow sizes, whereas the overhead of physical trailers can be significant, especially for applications with large flows and smaller packets. *ChainPatrol* is shown to provide a significant reduction in communication overhead (up to 45% of the original traffic) for common applications. Although the more complex virtual trailer design of *ChainPatrol* inevitably incurs slightly more end-to-end delay, such delay is still negligible for most applications (e.g., 20-30 ms is shown to be noticeable for interactive music, and 100 ms for games [31]). Finally, density reduction (physical) trailers can reduce the overhead, but it also causes the classification accuracy to deteriorate. Therefore, we conclude that virtual trailers enable *ChainPatrol* to provide a lightweight solution with both negligible communication overhead and negligible service delay, whereas physical trailers are better for applications with small flows with large packet sizes.

6.2.4 Handling unexpected jitters

In this experiment, we evaluate the two methods proposed in Section 3.4 for handling unexpected network jitters. As shown in Table 5, we evaluate our methods using artificial jitters up to six times the real-life jitter (i.e., the EC2 jitter in DS2[3]). The second and third columns show that both the X-shift method (for source VNF) and the α-amplitude method (for destination VNF) cause low false positive rates (up to 3% and 2%, respectively). The X-shift method shows a marginally higher rate, since it needs to update the X value upon every observation of unexpected jitters, and a delay in such updating may cause false positives. As to the α-amplitude method, it is "proactive" in the sense that using a larger-than-necessary amplitude may prevent false positives for multiple subsequent changes in jitters. The fourth and fifth columns of Table 5 show the end-to-end delay under only the X-shift method, and under both the X-shift and α-amplitude methods, respectively. We can see that both methods cause negligible increase in end-to-end delays, since they only introduce extra delays to one packet inside each pair, which affects the IPDs but does not significantly impact the end-to-end delay. Finally, Figure 11.F shows how the false positive rates is affected by the value of α under different jitters. Based on those results, *ChainPatrol* chooses the minimum α that can reduce the false positive rate to nearly zero (e.g., $\alpha = 1.5$ for $3x$ jitters).

[3] Note this experiment is performed several months later than those in Section 6.2.1, which explains the difference in delay (0.63 ms vs. 0.68 ms).

Table 5: The X-shift and α-amplitude methods

Δ jitter	X-shift FP(%)	α-amplitude FP(%)	X-shift delay(ms)	α-amplitude delay(ms)
0.04	2	1	0.63	0.65
0.08	3	1	0.64	0.66
0.12	3	2	0.64	0.66
0.16	2	1	0.65	0.67
0.20	3	2	0.65	0.67
0.24	2	2	0.66	0.68

7 Discussions

Security of *ChainPatrol*. The security of *ChainPatrol*, i.e., its capability of detecting and classifying attacks, depends on both the security guarantee provided by virtual trailers, and their successful decoding from IPD-based watermarks.

First, as virtual trailers are designed with similar fields as in their physical counterparts [6, 7] (detailed in Section 3.1), they can also provide similar cryptographic guarantees. First, as `FlowID` is uniquely and sequentially assigned to each flow, any injection, reordering, dropping, and replay of flows or VNFs can be easily detected using this trailer field. Second, `BlockNum` works similarly against those attacks at block level. Third, as it is computationally infeasible to forge a `MAC` (which is computed over both blocks and the previous two trailer fields) without knowing the secret key, packet-level attacks and tampering with virtual trailers themselves can both be detected using this trailer field. Finally, the continuous detection on all packets defeats the coward attack [9] (i.e., attacking only unprotected flows).

Second, virtual trailers encoded in IPD-based watermarks are much like physical trailers stored inside packets, which are both visible to adversaries (as confidentiality is not the goal), and can be easily modified or removed by them. Therefore, the issue of watermark invisibility (i.e., using smaller delays [17] or delays similar to normal network jitters [16] to hide watermarks against attacks that aim to locate the watermarks [18, 32]) is not applicable in our case, as we do not intend to hide the watermarks. Although adversaries can freely modify or remove watermarks in any way they like (by tampering with IPDs), the corresponding changes to virtual trailers can always be detected.

Specifically, adversaries can prevent the decoding of virtual trailers by changing all IPDs to equal values, randomly perturbing them, or replacing them with fake values. Although all those attacks will be detected by *ChainPatrol* (as failed decoding of virtual trailers indicates an attack), attack classification may become difficult or infeasible due to lack of virtual trailers. However, this essentially amounts to a denial of service attack (the same may happen to physical trailers, e.g., an adversary can simply delete them), which is out of the scope of this paper (a large amount of missing or destroyed trailers will likely trigger an investigation by the administrator). Finally, a smart adversary may disturb the IPDs in a way that mimics normal, but larger jitters to make the decoding harder. This will be handled by *ChainPatrol*,

as its agents continuously monitor the traffic and compensate observed jitters through both the encoding/decoding scheme (Section 3.2) and our X-shift approach (Section 3.4) (an unexplained increase in jitters will eventually raise suspicion from the administrator).

False negatives and positives. As mentioned above, virtual trailers share similar cryptographic guarantees with their physical counterparts, including no false negative and impossible-to-bypass [6, 7]. Note that, although virtual trailers and physical trailers can both be removed or modified by attackers, doing so will not cause false negatives as long as attackers cannot forge the MAC. On the other hand, a network disturbance such as out-of-order packets and packet loss can cause false positives under both virtual and physical trailers. What is unique to *ChainPatrol* is that jitters may also cause false positives (no effect on physical trailers), which has been addressed in our methodology (Section 3.4) and evaluation (Section 6.2.4), Also, as *ChainPatrol* agents are attached to two VNFs that directly communicate with each other, they are only affected by out-of-order or lost packets happening between them (which would be rare).

Limitations and future directions. The main limitations of *ChainPatrol* are as follows. First, *ChainPatrol* will report out-of-order packets or packet losses between two VNFs even if these are not caused by attacks. An interesting future work is to identify such cases (e.g, by learning different patterns of virtual trailers). Second, the limited capacity of side channel dictates less virtual trailers per packet, which causes slower attack detection (1.8s-2.4s detection time) and less accurate classification (around 70% accuracy). Whether such delays and accuracy are acceptable depends on use cases, e.g., reasonable for a regular IDS reporting attacks to humans (who can tolerate seconds-level delays and inaccuracies), but not ideal for real-time prevention (where physical trailer is better by working on each packet independently). A future direction is to improve those aspects through more compact design of virtual trailers. Finally, as a tenant-level solution, *ChainPatrol* cannot ensure the integrity of VNFs and cloud infrastructure like hardware-based solutions (e.g., [6, 12]).

8 Related work

SFC integrity. Traditional forwarding path verification protocols [8, 9, 33] cannot be directly applied to SFCs hosted in an NFV environment, since some of their underlying assumptions become unrealistic in the NFV context, e.g., forwarding paths are no longer fixed or known in advance in NFC, and network nodes (VNFs) are no longer transparent to packets as they might legitimately modify packets. More recent works tackle such issues to enable forwarding path verification in NFV. In [34], the authors study the issue of stateful and dynamic actions performed by VNFs, and propose a solution for VNFs to add tags to outgoing packets in order to bind packets with their origin. However, this scheme becomes ineffective when one or more switches are compromised. Therefore, FlowCloak [19] proposes an advanced packet tagging approach to randomize the tag generation such that the tags are probabilistically unknown by compromised switches. In contrast to *ChainPatrol*, FlowCloak requires modification to the internal logic of VNFs which may complicate its deployment. In [35], the authors propose a verification layer that is decoupled from the processing of VNFs, and is embedded in VMs supporting those VNFs. Nonetheless, vSFC requires modification at the cloud level, whereas *ChainPatrol* is a tenant-level solution that regards the cloud as a blackbox.

SFC-Checker [36] proposes a static analysis-based framework to ensure the correct behavior of dynamic and stateful forwarding paths. EasyOrch [37] performs verification based on a formal model that provides the flexibility of specifying both a forwarding policy and the set of anomalies to verify. In contrast to *ChainPatrol*, those solutions are static in nature and cannot detect run-time integrity breaches, as they either take snapshot of the network state to perform verification offline [36] or work before SFC deployment [37]. In [12], the authors propose a hardware-based solution to enclose both VNF processing and verification inside enclaves to preserve data confidentiality and VNF integrity against powerful adversaries, which are different from the focus of *ChainPatrol* (i.e., network links between VNFs). Closest to our work, AuditBox [6] provides runtime guarantees on the compliance with forwarding path policies using a hop-by-hop cryptographic trailer-based protocol, and by running each VNF inside enclaves. While *ChainPatrol* borrows similar design of trailers as AuditBox, our virtual trailer concept can significantly reduce its communication overhead (as demonstrated through experiments in Section 6).

Digital watermarking. There is a rich literature on digital watermarking in different contexts (e.g., image, audio, video, and network packets) and for different applications (e.g., copyright protection, traffic analysis, and tampering identification). The literature may be categorized along different dimensions. For instance, *blind* (e.g., [38]) or *non-blind* (e.g., [16, 17]) watermarking indicate whether the embedding and extraction of the watermarks require sharing knowledge about the original data. *ChainPatrol* leverages a blind watermarking scheme to avoid the overhead of sharing additional information about the original data. Second, for applications such as copyright protection and traffic analysis, the watermarks will be subject to either natural network noises on the Internet, or malicious tempering. Therefore, the watermarks should be *robust* [39] and/or *invisible* [16, 17] (i.e., hide watermarks against adversaries who aim to locate the watermarks [18, 32]). On the other hand, a *fragile* watermarking scheme (e.g., [40]) is mainly used for tampering identification as well as localisation of tampered data, and hence the watermarks should be sensitive to modifications and are not necessarily invisible. *ChainPa-*

trol belongs to the fragile (and non-invisible) category, since its watermarks are used to detect and classify tampering. The key innovation of *ChainPatrol* is it introduces an additional abstraction layer, i.e., virtual trailers, over watermarks, in order to leverage the latter's cryptographic properties.

Timing-based side channels are shown to have a larger capacity to share more information compared to other side channels [29]. In particular, packet timing-based flow watermarking modulates the IPDs of target network flow to embed watermarks and achieve the goal of linking flows for different applications, such as detection of stepping stone attacks, and compromising anonymity systems. For instance, the authors in [38] propose an IPD-based probabilistically robust watermarking scheme, which embeds watermark bits through slightly adjusting the independently and randomly selected IPDs. In [41], the authors propose an enhanced scheme where the watermarker can adaptively choose values of watermark parameters according to packet timing and packet size features of target flows. In [42], the authors propose a scheme that resists timing perturbations through grouping-based flow watermarking. In [43], the authors propose a flow watermarking technology based on packet matching and IPDs. In [44], the authors propose a blind flow watermarking system, which modulates fingerprints into the timing patterns of network flows through slightly delaying packets into secret time intervals only known to the fingerprinting parties. More recent works explore the intersection between machine learning and watermarking schemes. For instance, Fang et al. [45] apply deep learning techniques to obtain more robust flow-based watermarking schemes to ensure high consistency between the encoder and the decoder, and Xu et al. [46] design a watermarking scheme for graph data in order to verify the ownership of Graph Neural Networks (GNN) models.

9 Conclusion

Deploying network functions on top of existing cloud infrastructures makes it challenging for tenants to detect cloud-level attacks. Existing solutions based on cryptographic trailers can incur significant overhead for applications with small packets. In this paper, we proposed a novel concept, *virtual trailer*, which leveraged the inter-packet delay-based side-channel to encode cryptographic trailers without adding extra bits to packets. We developed *ChainPatrol*, a solution for encoding/decoding virtual trailers, and detecting and classifying various SFC attacks based on virtual trailers. We implemented and deployed *ChainPatrol* based on Amazon EC2, and our experimental results confirmed its effectiveness and efficiency.

Acknowledgment. We thank the anonymous shepherd and reviewers for their valuable comments. This work was supported by the Natural Sciences and Engineering Research Council of Canada and Ericsson Canada under the Industrial Research Chair in SDN/NFV Security and the Discovery Grant N01035, and by the Canada Foundation for Innovation under JELF Project 3859.

References

[1] ETSI, "Network functions virtualisation (NFV) release 3; management and orchestration; architecture enhancement for security management specification," 2018.

[2] P. Jiang, Q. Wang, M. Huang, C. Wang, Q. Li, C. Shen, and K. Ren, "Building in-the-cloud network functions: Security and privacy challenges," *Proceedings of the IEEE*, vol. 109, no. 12, pp. 1888–1919, 2021.

[3] Ammar Latif, Ash Khamas, Sundeep Goswami, Vara Prasad Talari, and Young Jung, "Telco meets AWS cloud: Deploying DISH's 5G network in AWS cloud," 2022, available at: https://aws.amazon.com/blogs/industries/telco-meets-aws-cloud-deploying-dishs-5g-network-in-aws-cloud/.

[4] VMware, "VMware expands its VMware ready for telco cloud program to accelerate the deployment of 5G services," 2020, available at:t.ly/BIIW.

[5] S. L. Thirunavukkarasu, M. Zhang, A. Oqaily, G. S. Chawla, L. Wang, M. Pourzandi, and M. Debbabi, "Modeling NFV deployment to identify the cross-level inconsistency vulnerabilities," in *CloudCom. IEEE.*, 2019.

[6] G. Liu, H. Sadok, A. Kohlbrenner, B. Parno, V. Sekar, and J. Sherry, "Don't yank my chain: Auditable NF service chaining," in *NSDI.USENIX.*, 2021.

[7] X. Zhang, Q. Li, Z. Zhang, J. Wu, and J. Yang, "vSFC: Generic and agile verification of service function chains in the cloud," *IEEE/ACM Transactions on Networking*, vol. 29, no. 1, pp. 78–91, 2020.

[8] J. Naous, M. Walfish, A. Nicolosi, D. Mazieres, M. Miller, and A. Seehra, "Verifying and enforcing network paths with ICING," in *Proceedings of the Seventh Conference on Emerging Networking Experiments and Technologies*, 2011, pp. 1–12.

[9] T. H.-J. Kim, C. Basescu, L. Jia, S. B. Lee, Y.-C. Hu, and A. Perrig, "Lightweight source authentication and path validation," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, 2014, pp. 271–282.

[10] M. Legner, T. Klenze, M. Wyss, C. Sprenger, and A. Perrig, "EPIC: Every packet is checked in the data plane of a path-aware internet," in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 541–558.

[11] M. Zoure, T. Ahmed, and L. Réveillére, "Network services anomalies in NFV: Survey, taxonomy, and verification methods," *IEEE Transactions on Network and Service Management*, 2022.

[12] S. Yao, M. Xu, Q. Li, J. Cao, and Q. Song, "cSFC: Building credible service function chain on the cloud," in *2019 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2019, pp. 1–6.

[13] N. Alhebaishi, L. Wang, and S. Jajodia, "Modeling and mitigating security threats in network functions virtualization (NFV)," in *IFIP DBSec*. Springer, 2020.

[14] V. Moorthy, R. Venkataraman, and T. R. Rao, "Security and privacy attacks during data communication in software defined mobile clouds," *Computer Communications*, 2020.

[15] H. Wang, H. Sayadi, A. Sasan, S. Rafatirad, and H. Homayoun, "Hybrid-shield: Accurate and efficient cross-layer countermeasure for run-time detection and mitigation of cache-based side-channel attacks," in *CCD*, 2020, pp. 1–9.

[16] A. Houmansadr and N. Borisov, "The need for flow fingerprints to link correlated network flows," in *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 2013, pp. 205–224.

[17] A. Houmansadr, N. Kiyavash, and N. Borisov, "Non-blind watermarking of network flows," *IEEE/ACM Transactions on Networking*, vol. 22, no. 4, pp. 1232–1244, 2013.

[18] N. Kiyavash, A. Houmansadr, and N. Borisov, "Multi-flow attacks against network flow watermarking schemes." in *USENIX security symposium*. Berkeley, CA, 2008, pp. 307–320.

[19] K. Bu, Y. Yang, Z. Guo, Y. Yang, X. Li, and S. Zhang, "Flowcloak: Defeating middlebox-bypass attacks in software-defined networking," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 396–404.

[20] M. Flittner, J. M. Scheuermann, and R. Bauer, "Chainguard: Controller-independent verification of service function chaining in cloud computing," in *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2017, pp. 1–7.

[21] M. S. Aslanpour, S. S. Gill, and A. N. Toosi, "Performance evaluation metrics for cloud, fog and edge computing: A review, taxonomy, benchmarks and standards for future research," *Internet of Things*, vol. 12, p. 100273, 2020.

[22] N. Agarwal, A. K. Singh, and P. K. Singh, "Survey of robust and imperceptible watermarking," *Multimedia Tools and Applications*, vol. 78, pp. 8603–8633, 2019.

[23] M. S. Niaz and G. Saake, "Merkle hash tree based techniques for data integrity of outsourced data," *GvD*, vol. 1366, pp. 66–71, 2015.

[24] Amazon AWS, "Common elasticache use cases and how elasticache can help," 2023, available at: https://docs.aws.amazon.com/AmazonElastiCache/latest/mem-ug/elasticache-use-cases.html.

[25] AWS, "Building lambda functions with python," 2023, available at: https://docs.aws.amazon.com/lambda/latest/dg/lambda-python.html.

[26] Netresec, "Publicly available PCAP files," 2023, available at: https://www.netresec.com/?page=PcapFiles.

[27] A. Habibi Lashkari., G. Draper Gil., M. S. I. Mamun., and A. A. Ghorbani., "Characterization of tor traffic using time based features," in *Proceedings of the 3rd International Conference on Information Systems Security and Privacy*. SciTePress, 2017, pp. 253–262.

[28] S. Jorgensen, J. Holodnak, J. Dempsey, K. d. Souza, A. Raghunath, V. Rivet, N. DeMoes, A. Alejos, and A. Wollaber, "Extensible machine learning for encrypted network traffic application labeling via uncertainty quantification," *IEEE Transactions on Artificial Intelligence*, pp. 1–15, 2023.

[29] A. Iacovazzi and Y. Elovici, "Network flow watermarking: A survey," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 1, pp. 512–530, 2016.

[30] H. Hawilo, M. Jammal, and A. Shami, "Exploring microservices as the architecture of choice for network function virtualization platforms," *IEEE Network*, vol. 33, no. 2, pp. 202–210, 2019.

[31] S. Liu, X. Xu, and M. Claypool, "A survey and taxonomy of latency compensation techniques for network computer games," *ACM Comput. Surv.*, vol. 54, no. 11s, sep 2022.

[32] Z. Lin and N. Hopper, "New attacks on timing-based network flow watermarks," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 381–396.

[33] IETF, "Proof of transit," 2020, available at: https://datatracker.ietf.org/doc/draft-ietf-sfc-proof-of-transit/.

[34] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 543–546.

[35] X. Zhang, Q. Li, J. Wu, and J. Yang, "Generic and agile service function chain verification on cloud," in *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*. IEEE, 2017, pp. 1–10.

[36] B. Tschaen, Y. Zhang, T. Benson, S. Banerjee, J. Lee, and J.-M. Kang, "SFC-Checker: Checking the correct forwarding behavior of service function chaining," in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2016, pp. 134–140.

[37] F. Valenza, S. Spinoso, and R. Sisto, "Formally specifying and checking policies and anomalies in service function chaining," *Journal of Network and Computer Applications*, vol. 146, p. 102419, 2019.

[38] X. Wang and D. S. Reeves, "Robust correlation of encrypted attack traffic through stepping stones by manipulation of interpacket delays," in *Proceedings of the 10th ACM conference on Computer and communications security*, 2003, pp. 20–29.

[39] P. Kadian, S. M. Arora, and N. Arora, "Robust digital watermarking techniques for copyright protection of digital data: A survey," *Wireless Personal Communications*, vol. 118, pp. 3225–3249, 2021.

[40] C. Qin, P. Ji, X. Zhang, J. Dong, and J. Wang, "Fragile image watermarking with pixel-wise recovery based on overlapping embedding strategy," *Signal processing*, vol. 138, pp. 280–293, 2017.

[41] Y. H. Park and D. S. Reeves, "Adaptive timing-based active watermarking for attack attribution through stepping stones," North Carolina State University. Dept. of Computer Science, Tech. Rep., 2007.

[42] Z. Pan, H. Peng, X. Long, C. Zhang, and Y. Wu, "A watermarking-based host correlation detection scheme," in *2009 International Conference on Management of e-Commerce and e-Government*. IEEE, 2009, pp. 493–497.

[43] P. Peng, P. Ning, D. S. Reeves, and X. Wang, "Active timing-based correlation of perturbed traffic flows with chaff packets," in *25th IEEE International Conference on Distributed Computing Systems Workshops*. IEEE, 2005, pp. 107–113.

[44] F. Rezaei and A. Houmansadr, "TagIt: Tagging network flows using blind fingerprints." *Proc. Priv. Enhancing Technol.*, vol. 2017, no. 4, pp. 290–307, 2017.

[45] H. Fang, Y. Qiu, K. Chen, J. Zhang, W. Zhang, and E.-C. Chang, "Flow-based robust watermarking with invertible noise layer for black-box distortions," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 4, 2023, pp. 5054–5061.

[46] J. Xu, S. Koffas, O. Ersoy, and S. Picek, "Watermarking graph neural networks based on backdoor attacks," in *2023 IEEE 8th European Symposium on Security and Privacy (EuroSP)*, 2023, pp. 1179–1197.

Appendix

Algorithm 1 details the generation of virtual trailers (applied by both the source and destination VNFs, for encoding and decoding, respectively). It takes as input a flow F and a SuperBlock SB in F. The algorithm first checks if F has already been assigned a `FlowID`, i.e., SB is not its first SuperBlock (Line 4), and if so the `FlowID` is reused and the `BlockNum` is incremented from the last assigned value (Lines 5-7). If F is a new flow (Line 8), then the `FlowID` is incremented from the last assigned value if the pair of source and destination VNFs is previously seen (Lines 9-10), or `FlowID` is randomly assigned otherwise (Lines 11-12). In both cases, the flow is marked as seen (Line 13), and the `BlockNum` is incremented from a random value (Lines 14-16). The MAC value is computed over the SuperBlock SB, if SB is not empty (Lines 17-18); otherwise, the MAC is filled with zero bits (Lines 19-20) (the latter case is to generate only the first two fields at the destination side). Finally, the virtual trailers are constructed based on aforementioned values (Lines 21-22).

Algorithm 1 Virtual Trailers Generation per SuperBlock

```
1:  Input: F: a flow; SB: a Superblock in F
2:  Output: VT[][]: virtual trailers of the SuperBlock
3:  procedure GENVT(SB,F)
4:      if (F ∈ F_ex) then                    ▷ Check if the flow (7-tuple) exists in list
5:          FlowID = getFlowID(F)
6:          for (i = 1..S_F) do
7:              BlockNum[i] = Inc(getLastBlockNum(F))
8:      else                                  ▷ 7-tuple not in list
9:          if (F.SrcVNF, F.DstVNF) ∈ F_ex then
10:             FlowID = Inc(getLastFlowID(F, F.destination))
11:         else
12:             FlowID = setFlowID(F)
13:         Add(F, FlowID, F_ex)              ▷ Add flow to list
14:         BlockNum[1] = setBlockNum(F)
15:         for (i = 2..S_F) do
16:             BlockNum[i] = Inc(getLastBlockNum(F))
17:     if isEmpty(SB) == false then
18:         MAC = MTHMAC(k, SB, BlockNum[], FlowID)
19:     else
20:         MAC = ZeroMAC()
21:     for (i = 1..S_F) do
22:         VT[i][] = <BlockNum[i], FlowID, SubSeq(MAC,i,S_F)>
23:     return VT[][]
```

Algorithm 2 details the encoding of the virtual trailers of a given SuperBlock. The input is the virtual trailers `VT[][]` generated by Algorithm 1, and the output is the ordered list of the new IPDs `EncodedVT[][]`. The algorithm iterates over each virtual trailer (Line 4-5), converts each frame into

Algorithm 4 Attack Detection

```
1: Input: F: a flow;
2: Output: F with each SuperBlock marked as recovered or attacked;
3: procedure DETECTATTACK(F)
4:     SB1 = NextSuperBlock(F, φ)                          ▷ 1st SuperBlock of F
5:     while (SB1 <> LastSuperBlock(F)) do
6:         SB2 = NextSuperBlock(F, SB1)
7:         DVT[][] = DECODEVT(SB2)                          ▷ Calling Alg. 3
8:         MAC1 = Concatenate(DVT[i][3]_{1≤i≤S_F})
9:                                  ▷ S_F: # of blocks in SuperBlock
10:        MAC2 = MHTMAC(k,SB1, DVT[i][1..2]_{1≤i≤S_F})
11:        if (MAC == MAC2 and DVT[i][1..2]_{1≤i≤S_F} match SB1)
   then
12:            MarkRecovered(F, SB1)
13:            SB1 = SB2                                    ▷ slide by a SuperBlock
14:        else                                             ▷ Attack Detected
15:            MarkAttacked(F,{SB1,SB2}) ▷ SB1 or SB2 or both attacked
16:            SB1 = SB1 + (2 packets)                      ▷ slide by two packets
17:    return F
```

Algorithm 2 Virtual Trailer Encoding

```
1: Input: VT[][]: Virtual trailers of SuperBlock SB with S_F blocks;
2: Output: EncodedVT[][]: New IPDs encoding VT[][];
3: procedure ENCODEVT(VT[][])
4:     for (i = 1...S_F) do
5:         for (i = j...3) do
6:             m[i][j] = ConvertBitSequence(VT[i][j])
7:             EncodedVT[i][j] = ComputeNewIPD(IPD_AVG, a, m[i][j])
8:     return EncodedVT[][]
```

the binary representation (Line 6), and computes the new IPD for each bit (as mentioned above) (Line 7).

Algorithm 3 details the decoding of the virtual trailers from a SuperBlock. The input is the SuperBlock `SB`. The output is the list of decoded virtual trailers `DecodedVT[][]`. The algorithm measures the IPDs in `SB` (Line 4) and divides the IPDs into blocks (Line 5). It then iterates over each block (Line 6), divides the block into frames (Line 7), computes the binary representation for each frame (Line 8-10), and converts the latter to a virtual trailer field (Line 11).

Algorithm 3 Virtual Trailer Decoding

```
1: Input: SB: SuberBlock;
2: Output: DecodedVT[][]: Integers corresponding to decoded VT[][];
3: procedure DECODEVT(SB)
4:     rIPD[] = measureIPD(SB)
5:     rIPD_B[] = DivideBlocks(rIPD[], S_F)
6:     for (each block B_i in SB) do
7:         rIPD_F[i][] = DivideIFrames(rIPD_B[i], 3)
8:         for (j = 1...3) do
9:             M_F^d[j] = ComputeEncodedBits( IPD_AVG, a, rIPD_F[i][j])
10:            m[j]= ConvertBit(M_F^d[j])
11:            DecodedVT[i][j]=getInteger(m[j])
12:    return DecodedVT[][]
```

Algorithm 4 details the attack detection performed at the ingress of a destination VNF. It takes as input a flow F, and outputs F with each SuperBlock marked as recovered or attacked. The algorithm iteratively slides a moving window over the next two consecutive SuperBlocks in F (Line 4-6). In each iteration, the algorithm first decodes the virtual trailer embedded in the second SuperBlock using Algorithm 3 (Line 7). It then extracts the `MAC` fields from the decoded virtual trailers, and concatenates them to obtain the decoded MAC value (Line 8-9). Then, it recomputes the Merkle tree MAC using all packets in $SB1$ together with the decoded `BlockNum` and the `FlowID` virtual trailer fields (Line 10). If those two values match and all the `BlockNum` and the `FlowID` virtual trailer fields match the first SuperBlock (Line 11), then the first SuperBlock is marked as recovered (Line 12), and the window slides forward by one complete SuperBlock (Line 13). Otherwise, there is an attack on at least one of those SuperBlocks (Line 14), so both SuperBlocks are marked as attacked (Line 15) and the window slides forward by two packets (Line 16).

Algorithm 5 outlines the attack classification. It takes as inputs the collection of flows marked by attack detection (Algorithm 4) and the pairs of attack types and their classification rules. It outputs the flows with each attacked SuperBlock classified as one of the attack types. The algorithm iterates over each flow (Line 4) and each attacked SuperBlock (Line 5), and applies each of the classification rules (Line 6). If the virtual trailers of the attacked SuperBlock can be successfully reconstructed (Line 7) under a rule, the SuperBlock is classified with the corresponding attack type (Line 8). If the virtual trailers can be partially reconstructed and the classification result can be verified by requesting additional information from the source (Line 9), then the SuperBlock is also classified with the attack type (Line 10).

Algorithm 5 Attack Classification

```
1: Input: 𝓕[] = {F}: M flows marked by attack detection (Algorithm 4);
   CR[] = {⟨A, R⟩}: N pairs of attack types and their classification rules;
2: Output: 𝓕 with attacked SuperBlocks classified by attack types;
3: procedure CLASSIFYATTACK(𝓕)
4:     for (i = 1...M) do
5:         while ((SB = NextAttackedSuperBlock(F_i)) <> φ) do
6:             for (j = 1...N) do
7:                 if ReconstructVT(SB, R[j]) == TRUE then   ▷ If VT of SB
   can be reconstructed with the j^th rule
8:                     Classify(SB, A[j])   ▷ Classifying SB as the j^th attack
   type
9:             else if (SrcAsstVerify(PartialReconVT(SB, R[j])) == TRUE)
   then                ▷ If partially reconstructed and verified with source
10:                Classify(SB, A[j])
11:    return 𝓕
```