# Symphony: Path Validation at Scale

Anxiao He, Zhejiang University, zjuhax@zju.edu.cn
Jiandong Fu, Zhejiang University, jiandongfu@zju.edu.cn
Kai Bu*, Zhejiang University, kaibu@zju.edu.cn
Ruiqi Zhou, Zhejiang University, rqzhou@zju.edu.cn
Chenlu Miao, Zhejiang University, clmiao@zju.edu.cn
Kui Ren, Zhejiang University, kuiren@zju.edu.cn

***Abstract*—Path validation has long been explored as a fundamental solution to secure future Internet architectures. It enables end-hosts to specify forwarding paths for their traffic and to verify whether the traffic follows the specified paths. In comparison with the current Internet architecture that keeps packet forwarding uncontrolled and transparent to end-hosts, path validation benefits end-hosts with flexibility, security, and privacy. The key design enforces routers to embed their credentials into cryptographic proofs in packet headers. Such proofs require sufficiently complex computation to guarantee unforgeability. This imposes an inevitable barrier on validation efficiency for a single packet. In this paper, we propose aggregate validation to implement path validation in a group-wise way. Amortizing overhead across packets in a group, aggregate validation promises higher validation efficiency without sacrificing security. We implement aggregation validation through Symphony, with various design techniques integrated and security properties formally proved. In comparison with state-of-the-art EPIC, Symphony speeds up packet processing by 3.78×∼18.40× and increases communication throughput by 1.13×∼6.11×.**

## I. Introduction

Path validation promises a fundamental solution to secure future Internet architectures. It is propelled by the Path-Aware Networking Research Group (PANRG) [27], [30] under the Internet Engineering Task Force (IETF) and Internet Research Task Force (IRTF). SCIONLab [15], [34]—a global research network—has already deployed path-validation–enabled routers into the Internet architecture for practice. In a nutshell, path validation secures packet forwarding with two complementary functionalities, path enforcement and path verification [7], [11]. Path enforcement specifies the designated path a packet should follow in the packet header. On-path routers are expected to adhere to the packet-carried path indicator for making forwarding decisions. However, malicious or compromised routers may mis-forward packets away from the specified path [9], [10], [33], [36], [39]. Such packet misforwardings tend to degrade performance and breach security [7]. Therefore, path validation further necessitates the other functionality—path verification. It verifies whether packet forwarding strictly follows the specified path. To this end, on-path routers embed unforgeable cryptographic proofs into the packet header. Such proofs should suffice for validating that a packet traverses not only all the on-path routers but also in the correct order [33].

Security and efficiency requirements impose an intrinsic dilemma on path validation solutions. Security requires long proofs and complex computation to make proofs hard to forge. However, efficiency requires short proofs and simple computation to control the overhead of packet transmission in links and packet processing on routers (Section VII). The dilemma between security and efficiency has already led to an efficiency bottleneck for the evolution of path validation solutions. No matter whether symmetric or asymmetric cryptography is used to compute proofs, the key length and associated proof size are lower-bounded given a required security level.

State-of-the-art path validation solutions have to trade off security for efficiency. For example, PPV [48] requires that packets be validated by only two randomly chosen adjacent routers rather than every router on the forwarding path. Each packet thus can testify its forwarding correctness on a path segment. A sufficient number of packets from the source to the destination can jointly reflect the forwarding behavior of the entire path. However, it does not satisfy the security requirement of hop-wise validation. To explore efficiency while maintaining hop-wise validation, the state-of-the-art EPIC [36] requires that routers embed only the leading bytes of their proofs in the packet header. Shorter proofs are more vulnerable to proof forging attacks [36]. Its hop-wise validation therefore leads to weakened security.

In this paper, we propose aggregate validation for efficient yet secure path validation. It validates path compliance during packet forwarding in a group-wise way. In contrast to existing packet-wise validation, aggregate validation aggregates a group of packets and uses them together as a single input for computing path proofs. The proofs are then distributed across packets in the same group. We can thus amortize validation overhead to improve efficiency yet without sacrificing security. We propose various techniques to achieve correct and efficient packet aggregation and implement them through Symphony. We also suggest a hardware queuing logic using static random-access memory (SRAM) for Symphony-enabled routers.

We implement Symphony using the Data Plane Development Kit (DPDK) [28]. We then carry out extensive experiments on both a local server and a multihop testbed using rented servers from Alibaba Cloud [17]. The experimental results show that Symphony can outperform the state-of-the-art EPIC [36] with faster packet processing and higher communication throughput. Specifically, the reduction of construction time ranges from 39.51% (Symphony with a 2-packet group) to 93.78% (Symphony with a 16-packet group). The processing speedup on routers increases to 18.40× when Symphony

*Kai Bu is the corresponding author.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.23104
www.ndss-symposium.org

with 16-packet groups is deployed. Symphony can achieve a 9.19 Gbps throughput given a single-core machine within a 10 Gbps link, being 6.11× higher than that of EPIC. We further measure the retransmission performance of Symphony given packet losses. Symphony can yield higher throughput than EPIC does even when the packet loss rate reaches a practical high like 10%.

In summary, we make the following major contributions to efficient path validation.

- Propose an aggregate validation technique and implement it through Symphony to fundamentally improve the efficiency of path validation without sacrificing security (Section III and Section IV). Aggregate validation yields not only shorter proofs per packet but also less queuing time per packet than existing packet-wise validation does.
- Model and validate Symphony throughput given packet losses (Section III and Section VI). Symphony can still outperform the state-of-the-art packet-wise path validation solution under practically high packet loss rates. To further prepare Symphony for disruptive network conditions with even higher packet loss rates, we explore a packet reaggregation technique that enables an intermediate router to function as a new source and re-initiate aggregation validation over the remaining verified packets of a group. Retransmission is then necessary for only lost packets instead of all the groups they pertain to.
- Prove the security and evaluate the performance of Symphony using the DPDK [28] on a local server as well as a multihop testbed (Section V and Section VI). In comparison with state-of-the-art EPIC [36], Symphony speeds up packet processing by 3.78×∼18.40× and increases communication throughput by 1.13×∼6.11×.

## II. Problem

In this section, we define path validation and review its performance requirements for security and efficiency. Related notations and abbreviations are defined in Table I. Security requirements enforce sufficiently complex computation such that path validation is robust against various attacks. However, this leads to an efficiency barrier that can hardly be overcome. State-of-the-art path validation solutions have to trade security for efficiency (Section VII).

### A. System Model

Path validation requires that the Internet architecture be incorporated with lightweight enhancements [33], [36], [39]. Specifically, these enhancements allow end-hosts to select forwarding paths and, together with on-path routers, to enforce and verify path compliance. First, the Internet architecture should be enhanced to authorize path selection requests from end-hosts. The authorization can rely on either an independent centralized authority [39] or collaborative distributed routers [36]. The granularity of the authorized paths has two levels. A fine-grained path consists of all the routers across different Autonomous Systems (ASes). It takes into account both intra-AS and inter-AS forwarding reliability. In contrast, a coarse-grained path abstracts each AS into a single node and focuses on only inter-AS forwarding [36]. Different path granularities impose no significant impact on the design of path validation solutions. We follow the commonly adopted inter-AS path model as in the literature [33], [36]. Second, routers need to update the packet process logic in favor of packet headers with embedded path proofs. Finally, end-hosts per se need also to update protocol stacks for requesting paths and processing packets (e.g., encapsulation and parsing).

TABLE I. Definition of notations and abbreviations.

| Notation | Definition |
|---|---|
| $\Phi$ | path consisting of routers $(R_0, ..., R_i, ..., R_n)$ |
| $n$ | length of path $(R_0, ..., R_i, ..., R_n)$ |
| $S$ | source node, that is, $R_0$ |
| $D$ | destination node, that is, $R_n$ |
| $R_i$ | on-path router for $i \in [1, n-1]$ |
| $K_i$ | shared symmetric key between $R_0$ and $R_i$ |
| $G$ | the group of packets |
| $m$ | size of a packet group, that is, the number of packets in the group |
| $P_i$ | the $i$th packet of a group, where $i \in [1, m]$ |
| $GP$ | group payload concatenated by all the payloads in a group |
| $\sigma$ | proof field for $R_i$ to compute proof |
| $V_i$ | proof field for $R_i$ to verify source and path |
| $H(\cdot)$ | cryptographic hash function |
| $\text{MAC}_K(\cdot)$ | message authentication code (MAC) using key $K$ |
| $\|\|$ | concatenation of strings |
| $\Psi(G, R_i)$ | validation function on router $R_i$ over packet group $G$ |
| $r$ | state of received packets on routers |
| $s$ | state of sent packets on routers |

### B. Threat Model

Path validation defends against an active Dolev-Yao attacker [19] that conducts packet mis-forwardings while trying to forge valid path proofs [33], [36], [39]. The attacker can be a malicious router per se or a router hijacked by the attacker. A packet mis-forwarding occurs whenever the attacker directs a packet without strictly following its specified path. For example, the attacker may sidestep a certain router to evade its connected packet inspection service. Moreover, the attacker may direct packets across on-path routers in a different order than is required. This also tends to breach security policies [8]. To make a mis-forwarded packet pass path validation, the attacker intends to forge its path proofs such that the forged proofs are deemed valid on routers or end-hosts.

### C. Security Requirements

At the heart of path validation lies two critical security requirements, proof unforgeability and hop-wise validation. Proof unforgeability is a must for security and correctness of path validation. Only when path proofs are hard to forge can mis-forwarded packets be detected and filtered. Hop-wise validation aims to filter mis-forwarded packets as soon as possible instead of postponing packet filtering to the destination. This helps to minimize security impacts and resource wastes that might be caused by in-network mis-forwarded packets.

Following a commonly used network-security formal-proof method called Protocol Composition Logic (PCL) [42], [50], we define the predicate $(\text{goodV}\ n\ S\ D\ path\ pkt\ key_1\ key_2\ t)$ to indicate that a packet with payload $pkt$ has passed verification using key $key_1$ and updated the proof using key $key_2$ in the $n$th router on a path $path$ with source $S$ and destination $D$ at time $t$. Table II defines other related predicates.

TABLE II. DEFINITION OF PREDICATES IN FORWARDING.

| Predicate | Definition |
|---|---|
| honest $R$ | $R$ is a trusted node of either the source, destination, or intermediate on-path routers |
| owner $i$ $rt$ | router $rt$ owns the thread $i$ responsible for processing the packet |
| pathT $\Phi$ $R$ $n$ | router $R$ is the $n$th hop on path $\Phi$ |
| Session $S$ $D$ $\Phi$ $keys$ | $S$ and $D$ have agreed on a session to forward packets along path $\Phi$ with related router keys $keys$ |
| recv $i$ $rt$ $(S, D, \Phi, pkt, proof_r)$ $tr$ | router $rt$ has received the packet that consists of payload $pkt$ and proof header $proof_r$ following the path $\Phi$ at time $tr$ in thread $i$ |
| send $i$ $rt$ $(S, D, \Phi, pkt, proof_s)$ $ts$ | router $rt$ has sent the packet that consists of payload $pkt$ and proof header $proof_s$ following the path $\Phi$ at time $ts$ in thread $i$ |
| acceptRt $i$ $pkt$ $proof_r$ $rt'$ $key$ $ta$ | thread $i$ has accepted the packet that consists of payload $pkt$ and proof header $proof_r$ using key $key$ at time $ta$ |

$$\frac{}{\text{goodV } 0\ S\ D\ \Phi\ pkt\ \sim\ K_1\ t}$$

$$\text{honest } R \quad \text{pathT } \Phi\ R\ n+1 \quad \text{owner } i\ R$$
$$\text{if } n = 0 \text{ then } rt' = S$$
$$\text{if } n = n' + 1 \text{ then pathT } \Phi\ rt'\ n$$
$$\text{recv } i\ rt'\ (S', D', \Phi', pkt, proof_r)\ tr$$
$$\text{send } i\ rt'\ (S', D', \Phi', pkt, proof_s)\ ts$$
$$tr < t \quad ts < t \quad tr < ts$$
$$\frac{\text{goodV } n\ S\ D\ path\ pkt\ K_{n-1}\ K_n\ t}{\text{goodV } n+1\ S\ D\ path\ pkt\ K_n\ K_{n+1}\ t}$$

In the base case when $n = 0$, it indicates no requirements for the source. Otherwise, suppose that $R_j$ and $R_{j+1}$ are the $j$th and $(j+1)$th routers on path $\Phi$. If $R_{j+1}$ is honest, there exist two times—$tr$ and $ts$—that router $R_{j+1}$ receives the packet from $R_j$ at time $tr$ and sends the packet out at time $ts$. The additional requirement is that the behavior on $R_j$ should be correct. If $R_{j+1}$ is dishonest, it behaves according to the threat model defined in Section II-B and no requirement is enforced on it.

**Requirement 1.** *The security requirement of path validation:*

$$\forall S, D, \Phi, rt, pkt, keys, proof_r, ta, i, rt',\ \text{s.t.}:$$
$$\text{Session } S\ D\ \Phi\ keys$$
$$\text{honest } S, \text{honest } rt$$
$$\text{owner } i\ rt \quad \text{acceptRt } i\ pkt\ proof_r\ rt'\ K_{n-1}\ ta$$
$$\exists n, j, ms, ts, proof_s,\ \text{s.t.}:$$
$$\text{goodV } n\ S\ D\ path\ pkt\ K_{n-1}\ K_n\ ta$$
$$ts < ta \quad \text{owner } j\ S$$
$$proof_s = \text{macMsg } K_n\ proof_r$$
$$\text{send } j\ rt'\ (S', D', \Phi', pkt, proof_s)\ ts$$

Requirement 1 states the basic procedure of path validation. If a router accepts a packet, then the packet must have a source-originated payload $pkt$ and has traversed all the source-designated honest routers in the right order with correctly verified and updated proofs. This way, Requirement 1 indicates exactly the two aspects of security requirements: proof unforgeability and hop-wise validation.

- **Proof unforgeability**. Since path validation uses path proofs to encode packet forwarding trajectories, proof construction schemes should be sufficiently secure. Specifically, the attacker forges as many proofs as possible in the hope that one of them can pass verification [36]. A common countermeasure should enforce a sufficiently high security level.
- **Hop-wise validation.** A mis-forwarded packet may associate with security risks and should not wander in the network or reach the destination. Hop-wise validation requires each on-path router to verify the path compliance on all its upstream routers. Once a mis-forwarded packet is detected, it should be immediately discarded. This also helps to economize network resources in that if mis-forwarded packets are not discarded, they will keep consuming network resources such as link bandwidth and router storage.

**Non-security goals.** Path validation does not aim to address the following security concerns [39]. The attacker copies packets and may further send these copies somewhere else for traffic monitoring and correlation. The attacker modifies packets. This is orthogonal to path validation and can be easily mitigated by integrity check. The attacker does not execute promised processing services on received packets other than simply forwarding them. This is also orthogonal to path validation. A feasible countermeasure uses probe packets with predictable processing results for attestation [7]. The attacker may even drop packets. Dropped packets can be retrieved by retransmission. Finally, path validation considers only the security of the data plane rather than that of the control plane [36], [47]. It assumes that forwarding paths dictated by the control plane are trusted and verifies whether the actual forwarding path complies with the dictated one. It is secure routing that guarantees the trustworthiness of path computation on the control plane [32], [38], [45]. Secure routing is orthogonal to secure forwarding [7], [47] using path validation and is beyond the scope of this paper.

### D. Efficiency Requirements

There are two ultimate efficiency requirements for network communication augmented with path validation—throughput and goodput [33], [36]. Throughput reflects how many packets can be transmitted in a certain time period. Goodput reflects how many payloads can be encapsulated in these packets. A high communication efficiency expects both throughput and goodput to be high.

**Throughput.** It quantifies the bandwidth consumed for transmitting packets augmented with path proofs [33].

**Goodput.** It quantifies the bandwidth consumed for transmitting packet payloads, which are the essential ingredient in data transmission. Given a fixed throughput, a higher ratio of the packet payload to the packet length yields a higher goodput. Therefore, we require that the proof size be sufficiently small to gain a high payload ratio.

## III. OVERVIEW

In this section, we explore an aggregate validation technique to fundamentally improve the efficiency of path validation without sacrificing security. It advocates group-wise

validation and takes a group of packets as a single input for computing the path proof. The proof is then evenly distributed across these packets. The aggregate validation technique yields not only shorter proofs per packet but also less queuing time per packet than existing packet-wise validation does. We implement it through Symphony, highlighting key strategies in this section and presenting design details in Section IV.

### A. Aggregate Validation

We explore the aggregate validation technique toward efficient and secure path validation. Unlike existing packet-wise validation, it improves efficiency by simultaneously validating a group of packets. We observe that the efficiency barrier over existing path validation solutions is mainly because they gradually improve efficiency from the perspective of an individual packet. No matter how the associative cryptographic schemes can be optimized, they have to be sufficiently secure against various attacks. This enforces an inevitable lower bound of overhead for embedding sufficiently long path proofs in packet headers and for performing sufficiently complex computation for generating, verifying, and updating the proofs. The efficiency barrier induced by packet-wise validation motivates us to explore the other perspective—aggregate validation. It aggregates a group of packets as an independent packet, which is then taken as the input of our path validation protocol. The proof and the associative computation cost are evenly amortized over these packets. This promises a shorter proof and faster validation per packet, exactly what a higher efficiency requires. Furthermore, by simply treating a group of packets as an aggregate packet, aggregate validation does not weaken any security constraint.

Shorter proofs supported by aggregate validation improve not only efficiency but also practicality of path validation. Proof size continues to be a major optimization target. Proof fields of traditional solutions using symmetric key encryption increase with path length. They tend to take too much space in the packet header and squeeze the space for the payload. For example, ICING [39] requires 42 more bytes per hop for a proof. Consider the 1,500-byte maximum transmission unit (MTU) limit [26]. It only takes 11 hops to run out of the space while the median length of network paths is nearly 16 hops [40]. Note that simply extending the MTU is not a fundamental fix. Once the MTU is set to whichever value, it becomes the new upper bound that limits the maximum path length a validation solution can support. The MTU cannot be set too large either. A moderate scale is vital for agile scheduling among connections that multiplex the same link [37]. EPIC [36] explores a workaround to greatly reduce the per-hop proof size to five bytes, which is taken from the entire proof and vulnerable to brute-force attacks. It does not shorten the proof for the destination such that any undetected invalid proofs on routers can be detected on the destination. This may not satisfy the hop-wise validation requirement. In contrast, our aggregate validation technique achieves shorter proofs by jointly validating a group of packets as one. The essential benefit of using packet groups is to amortize validation overhead to improve efficiency. This offers much speedup than traditional solutions that validate packets individually (regardless of whether the MTU of each packet is extended). Furthermore, distributing proofs across packets in the same group can save packet space

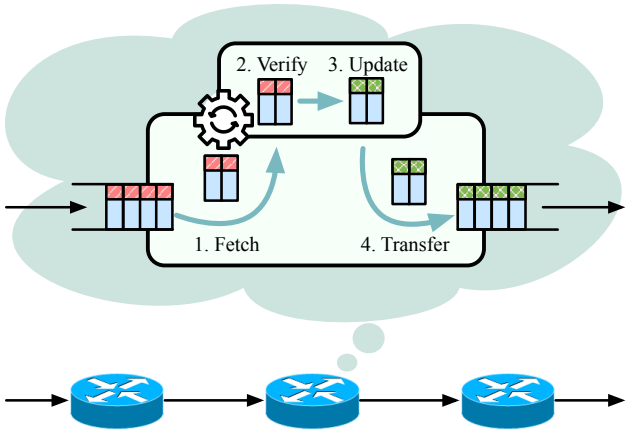



Fig. 1. Symphony architecture with aggregate path validation. Routers process packets in a group-wise way.

and, in turn, improve throughput. Aggregate validation thus guarantees secure path validation in a practically efficient way.

The feasibility of aggregate validation can be guaranteed by the high transmission rate of network traffic. Intuitively, if the transmission rate is low, a router may wait a while until sufficient packets constituting a group arrive. If the waiting time is too long, it may outweigh the aforementioned efficiency improvements that path validation offers. Fortunately, the transmission rate in modern networks is sufficiently high. It is practical that router queues may be constantly filled with a large number of packets [6]. This makes it feasible for a router to fetch a group of queued packets without having to wait as when insufficient packets are queued. Furthermore, path validation introduces additional operations than traditional packet processing does. This intrinsically makes even more packets than usual to be queued on routers. In other words, path validation makes packets stay a longer time in queues. Existing solutions shorten this waiting time only by trying to speed up the validation operations on each individual packet. In contrast, the aggregation validation technique we propose simultaneously fetches a group of packets from the queue and directly removes the waiting time interleaved among them.

### B. Symphony Protocol

We propose Symphony, the first path validation protocol that leverages aggregate validation to improve efficiency with guaranteed security. Figure 1 illustrates the framework of Symphony using a 2-packet group for example. At a high level, the packet processing logic of Symphony on routers revolves around the following four major steps.

**Step 1:** The router fetches a group of packets from the input queue.

**Step 2:** The router takes the entire group of packets as a single input and computes path validation proofs. Then it extracts the carried proofs from all packets and concatenates them. Finally, the router verifies proof validity by comparing the computed proof with the concatenated one. If they match, then the entire group of packets passes validation.

**Step 3:** If validation succeeds, the router further updates the packet proofs with its own credential integrated. The router still takes the entire group as a single input for proof computation.

The computed proof is evenly distributed across the packets of the same group, replacing the original proof.

**Step 4:** Finally, the router places the group of verified packets that carry updated proofs in the output queue.

**Router-queue logics.** In comparison with traditional validation per individual packet, Symphony using aggregate validation requires that router-queue logics support grouping and sorting operations over queued packets. Implementing such logics would be a hardware design question of various possible choices. We hereby suggest an SRAM-based solution and leave the exploration of other design choices to the practitioners of our Symphony protocol. Specifically, the original first in, first out (FIFO) queue on a router stores a series of packets mixed from different groups. Queued packets of the same group may possibly be out of order. We suggest mounting a small SRAM to facilitate grouping and sorting packets from the queue. SRAM chips feature fast and random accesses. To support Symphony, the mounted SRAM features a $g \times m$ grid of cells, where $g$ denotes the number of groups the queue can hold and $m$ denotes the number of packets per group. Each SRAM cell stores validation metadata of a queued packet. The ultimate goal is to fetch one packet after another from the FIFO queue and place them in a group-wise ordered fashion to the SRAM. Each group (indexed with GroupTag in Section IV-B) is assigned to a specific SRAM row. Within each row, metadata of a packet in this group (indexed with OrderTag in Section IV-B) are stored in the cell with the column index being equal to OrderTag, which tracks the order of packets in a certain group. This way, both grouping and sorting are addressed.

### C. Packet Aggregation

A major challenge for packet aggregation is to correctly identify packets of the same group and preserve the order used by the source for computing proofs. Without careful design, these two requirements may not be guaranteed. Given the high transmission rate and intricate network topology, a router constantly receives a large volume of packets from various input ports. This easily induces inter-group packet mixing on the router. Furthermore, it is normal that multi-core routers perform parallel processing on packets. Packet A arriving in the input queue earlier than packet B may reach the output queue later than packet B. This leads to intra-group packet disordering on enroute routers.

We introduce hierarchical tags in packet headers to address the challenges of inter-group mixing and intra-group disordering (Section IV-B). Note that all such header fields and those to be used hereafter belong to the Symphony header we introduce between the Internet Protocol (IP) and Transmission Control Protocol (TCP) headers (Section IV-A). The Symphony header consists of all the fields for our Symphony to compute and verify path proofs. It is a convention in existing solutions to place such path-validation proof fields between the IP and TCP headers [24], [33], [36], [39]. This renders path-validation implementation feasible for modular design and simplifies the upgrade of protocol stacks for packet processing on routers.

Now let us focus on how our introduced fields of hierarchical tags deal with inter-group mixing and intra-group disordering in particular. First, we assign packets of the same group with an identical group tag. Second, for same-group packets, we further assign them with different order tags. Consider a 10-packet group for example. The order tags for packets in this group can be assigned with values ranging from 0 to 9. Order tags can be reused across different groups. However, we should use group tags together with the source and destination IP addresses for uniquely classifying packet groups. More specifically, path validation solutions introduce a SessionID header field to specify the binding of two communication end-hosts and the forwarding path designated to them in a session. We thus use SessionID and group tags in the Symphony header to differentiate groups. Because different sources can hardly coordinate, it is challenging to count on them to guarantee the uniqueness of group tags. Group tags from different sources are possibly collided. Such collisions lead to inter-group packet mixing as well. We address this issue by a two-step packet classification method. The first step classifies packets according to their SessionID. The second step classifies packets using their group tags. For the same source-destination connection, we can reuse group tags if their associated packets have a sufficient time interval and do not overlap on routers.

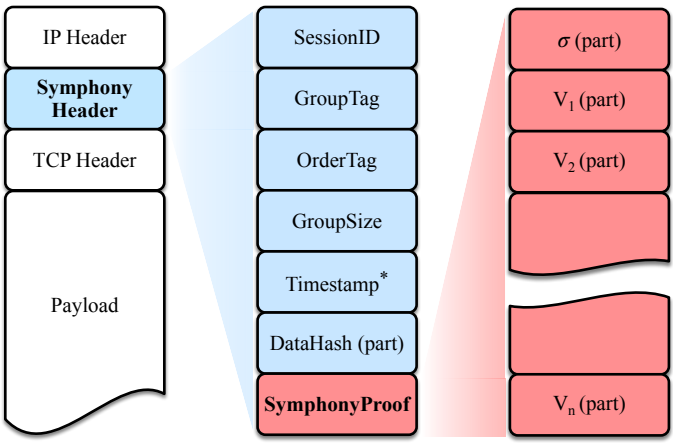



Fig. 2. Symphony header structure. Notations '*' and 'part' mean that the corresponding field is entirely stored in the first packet of a group and evenly divided into each packet of a group, respectively.

Furthermore, packet losses impose a potentially tougher challenge on Symphony than on existing solutions because one lost packet fails verification of an entire group and thus necessitates retransmission of the entire group. This might also lead to network congestion. We follow the TCP Bottleneck Bandwidth and Round-trip propgation time (BBR) model [13], [14], [46] to quantify the impacet of packet loss rates (Section VI-J). Symphony can still yield higher throughput than the state-of-the-art does given practically high packet loss rates.

## IV. Design

In this section, we detail Symphony design atop the aggregate validation technique.

### A. Header Structure

As with existing path validation solutions [24], [33], [36], [39], we place a Symphony header between the IP header and TCP header (Figure 2). It consists of seven fields—GroupTag, OrderTag, GroupSize, SessionID, Timestamp, DataHash, and

SymphonyProof. These fields fall into five categories with respective purposes. SessionID specifies a communication session between the source and destination. It also serves as a path indicator, dictating routers on which path to follow for a session of communication traffic. GroupTag, OrderTag, and GroupSize cluster and order packets in the same group of a certain session. Timestamp demonstrates packet freshness. DataHash protects packet integrity. SymphonyProof encapsulates the cryptographic path proofs constructed by Symphony.

**SessionID:** It is negotiated by the source and destination to track traffic in a communication session. In path validation, the source and destination can negotiate a path they prefer to follow for each session [33]. We carry SessionID in the packet header while storing it and the corresponding path information on routers. This not only saves header space but also improves processing efficiency. Against the rerouting attack, we use SessionID as an input for computing SymphonyProof.

**GroupTag, OrderTag, and GroupSize:** These fields are initialized at the source (Section IV-B). GroupTag specifies the group that a packet belongs to. OrderTag specifies the ordering index of the packet in the group. GroupSize specifies the number of packets in a group. It can be omitted if the group size is globally fixed. However, different sources may have different preferences. The same source may also have different preferences under different connection status or communication applications. Therefore, we explicitly include GroupSize in the header as an adjustable parameter. This facilitates routers to collect the correct number of packets for a group on the fly. Furthermore, upon a packet loss, GroupSize can help routers to detect whether a group of packets is complete on the fly.

**Timestamp:** It records the time when a group of packets is generated at the source and reflects packet freshness. When a router receives a packet, it needs to make sure that its Timestamp be reasonably recent. Timestamp also helps to prevent replay attacks [33] and should be fed into the computation of SymphonyProof. Timestamp is stored only in the first packet of a group as it is shorter than other commonly used fields such as SessionID and DataHash.

**DataHash:** It aims to support integrity check in a group-wise way. Specifically, we concatenate the payloads of all packets in a group to form a group payload $GP$. We then calculate DataHash—$H(GP)$—using a hash function $H(\cdot)$ that is shared by all end-hosts and routers. To prevent malicious routers from modifying payloads and forging DataHash, we require that DataHash be computed into SymphonyProof as well. We distribute DataHash evenly across the packets that are used to compute it instead of stacking it entirely to the first packet. We merge them together upon verification. Such a group DataHash promises a higher goodput than existing solutions with packet-wise hash do. Furthermore, it increases the difficulty for the attacker to forge payloads to satisfy a given DataHash because DataHash depends on a group of packets instead of only one.

**SymphonyProof:** As the key building block of a Symphony header, it is the only cryptography-protected field. SymphonyProof further consists of two types of fields, update field $\sigma$ and verification field $V_i$ (including $V_D$). Both types of fields are initialized by the source (Section IV-C). $V_i$ is fixed during packet transmission and serves as an assurance of forwarding packets along the designated path. In contrast, $\sigma$ is used to compute $V_i$ per hop and thus needs to be updated by intermediate routers. The computation also involves the aforementioned SessionID, Timestamp, and DataHash (Algorithm 1). We distribute both $\sigma$ and $V_i$ across packets.

### *B. Packet Aggregation*

Aggregate validation depends on correct and efficient packet aggregation. Once the source computes path proofs over an ordering group of packets, it is critical for routers and the destination to collect the same group of packets in the same order to pass validation. We introduce three header fields to support correct and efficient packet aggregation—GroupTag, OrderTag, and GroupSize. GroupTag and OrderTag specify the group that a packet belongs to and the ordering index in the group that a packet associates with, respectively. GroupSize following GroupTag and OrderTag explicitly specifies the number of packets in the group. This helps routers and end-hosts to determine how many packets to collect for a group, especially when GroupSize is an adjustable system parameter.

**GroupTag.** Each group should be uniquely identified against inter-group mixing. To explore an efficient way for uniquely identifying groups, we observe that the fields of source IP address, destination IP address, and SessionID can already uniquely identify a session. What we further need is to introduce GroupTag that can differentiate groups within a session. Moreover, SessionID is 16 bytes and has a negligible collision probability of $1/2^{128}$. It alone suffices to uniquely identify a session. Therefore, we can use SessionID and the newly introduced GroupTag to uniquely identify groups.

It is critical to set an appropriate length of GroupTag to avoid GroupTag collision in the same session. We classify groups using SessionID and GroupTag. If GroupTag is too short, the source will quickly reuse earlier GroupTag. A router may incorrectly classify groups when its queue contains packets from different groups yet with the same SessionID and GroupTag. To avoid such a collision, we require that when a packet with a reused GroupTag reaches the router next to the source, any packet with the same GroupTag should have already been processed and dequeued from the router. Let GroupTag and OrderTag be $t$ bits and $s$ bits, respectively. The maximum number of packets without group collisions is $2^{t+s}$. Consider an extra packet following the $2^{t+s}$ packets. The $2^{t+s}+1$ packets have a GroupTag collision between the extra packet and the first group in the packet stream. To avoid this collision, we need to satisfy the constraint:

$$\frac{(2^{t+s}+1)\times \mathrm{delay}_{\mathrm{transmission}}}{2^{s}\times \mathrm{delay}_{\mathrm{processing}}} > 1, \tag{1}$$

where $\mathrm{delay}_{\mathrm{transmission}}$ denotes the time span from the first bit of a packet reaches the router to the ending bit of the packet is queued and $\mathrm{delay}_{\mathrm{processing}}$ denotes the time for the packet to be processed by the router. As will be discussed shortly, GroupSize of 8 bits suffices to satisfy practical queue lengths on routers. Together with the measured transmission delay and processing delay in Section VI, we observe that GroupTag of 8 bits suffices to avoid group collisions in the same session.

**OrderTag.** Packets of the same group should also be ordered exactly the same as the source follows to compute path proofs.

TABLE III. EXAMPLE OF PACKET AGGREGATION.

| Packet | SessionID | GroupTag | OrderTag | GroupSize |
|---|---|---|---|---|
| A | $S1$ | 1 | 0 | 3 |
| B | $S1$ | 1 | 2 | 3 |
| C | $S2$ | 2 | 2 | 4 |
| D | $S1$ | 1 | 1 | 3 |

GroupTag supports packet ordering by explicitly specifying the ordering index of a packet in a group. We set the length of GroupTag to 8 bits considering the practical constraint of queue size on routers. If a group has too many packets, it would occupy excessive queue space and leave little room for other groups. In this case, the incoming packets might be dropped due to the drop tail effect [6]. This induces packet retransmission and corresponding overhead to end-hosts and routers. Given that the commonly used limit for queue size on routers is 140 [22], an 8-bit OrderTag can support up to 256 packets per group and suffices to record different packets.

**GroupSize.** We further augment GroupTag and OrderTag with GroupSize to instruct routers about the exact number of packets to collect for a group. It shares the same length with that of GroupTag such that it can support the maximum value of OrderTag. A major benefit of using an explicit GroupSize is to make group size an adjustable parameter. In comparison with a globally fixed group size, an adjustable one enables sources to choose the group size that can best suit its requirements for service and performance quality.

**Example of packet aggregation.** For ease of understanding, Table III shows an example of how a router uses the three fields—GroupTag, OrderTag, and GroupTag—together with SessionID to identify packets of the same group in the correct order out of out-of-order packets. We consider four packets belonging to two groups, where packets A, B, and D form a complete group. After the router identifies packet A, it classifies its group using GroupTag of 1 and SessionID of $S1$. Since packet A associates with OrderTag of 0 and GroupSize of 3, the router knows that it is the first packet of a group of 3 and two more packets await to complete the group. The router then meets packet B with the same SessionID and GroupTag with that of packet A. However, a disorder arises as OrderTag of packet B is 2 instead of 1 that follows 0 of packet A. OrderTags help the router to notice the packet disorder and GroupSize helps the router to wait for one more packet from this group. Then it comes packet C that has different GroupTag and SessionID from that of packets A and B. The router omits packet C and proceeds to identify subsequent packets in the queue. Finally, the router identifies that packet D has the same SessionID and GroupTag with that of packets A and B. Packet D thus brings forward the third packet to complete the group. Using OrderTags of packets A, B, and D, the router orders them in the sequence of A, D, and B.

### C. Aggregate Validation

As shown in Algorithm 1, aggregate validation enables three types of entities (i.e., source, router, and destination) to run four types of functions (i.e., `Initialization`, `Construction`, `Verification`, and `Update`). For initialization, the source exchanges a respective shared symmetric key with every router and the destination. These keys are used for proof construction, verification, and update. Once a specific group of packets is ready, the source initializes their GroupTag, OrderTag, and GroupSize for packet aggregation. The source then constructs group proofs over the group of packets and sends these packets to the next hop. Upon receiving the group of packets, a router verifies their proofs. If verification succeeds, the router updates the proofs by integrating its own credentials and forwards the group of packets to the next hop. The preceding proof verification and update iterate until the group of packets arrive at the destination. Since the destination has no next hop to forward packets to, it only verifies proofs rather than updates them.

**Algorithm 1: Symphony – Aggregate Validation**

**Function** `Initialization`**:**
  **for** $i \in [1, n]$ **do**
    $R_0$ and $R_i$ exchange shared symmetric key $K_i$;

**Function** `Construction`**:**
  **for** *packet* $P_i \in$ *a group of* $m$ *packets* **do**
    SessionID $\leftarrow$ identifier of the current session;
    Generate GroupTag, OrderTag, and GroupSize;
  Timestamp $\leftarrow$ creation time of packet $P_1$;
  Embed Timestamp in the header of only $P_1$;
  $GP \leftarrow P_1.\text{payload}||...||P_m.\text{payload}$;
  DataHash $\leftarrow H(GP)$;
  Divide DataHash into all packet headers;
  //compute SymphonyProof;
  $h \leftarrow$ SessionID||DataHash||Timestamp;
  $\sigma_0 \leftarrow \text{MAC}_{K_n}(h)$;
  $\sigma \leftarrow \sigma_0$;
  Divide $\sigma$ into all packet headers;
  **for** $i \in [1, n]$ **do**
    $V_i \leftarrow \text{MAC}_{K_i}(h||\sigma_{i-1})$;
    Divide $V_i$ into all the packets;
    $\sigma_i \leftarrow \text{MAC}_{K_i}(\sigma_{i-1})$;

**Function** `Verification`**:**
  **if** *a group is complete* **then**
    $h \leftarrow$ SessionID||DataHash||Timestamp;
    $V_i' \leftarrow \text{MAC}_{K_i}(h||\sigma)$;
    **if** $V_i == V_i'$ **then**
      Accept the group of packets;
    **else**
      Drop the group of packets;
  **else**
    Drop the group of packets;

**Function** `Update`**:**
  //If verification succeeds, router $R_i$ updates $\sigma$;
  $\sigma \leftarrow \text{MAC}_{K_i}(\sigma)$;

Prior to detailing the design of each function, we further generalize the design guidelines (especially about proofs) for ease of understanding. One can imagine proofs as a series of pre-computed integrity measures. First, they resemble integrity measures in that a node (i.e., an intermediate router or the destination) should know the exact expected proofs after all its upstream nodes (i.e., the source or intermediate routers) follow the designated forwarding order and correctly update the proofs in the packet header. Second, proofs can be pre-

computed and then pre-arranged in the packet header because the source is usually deemed trusted in practice [9], [10], [33], [36], [48]. Specifically, the trusted source needs to share with each co-path node a secret key. A co-path node will use its shared key to integrate its credentials into the proofs. Since the shared key is also known to the source, it is possible for the source to pre-compute the expected proof after a packet has traversed certain hops. Proofs may vary hop by hop. The source then embeds these pre-computed proofs in the header of an outgoing packet. Upon a subsequent node receives the packet, it essentially runs an integrity check by 1) computing the expected proof with its shared key and packet metadata as well as 2) comparing the computed proof with the proof that is carried in the packet header. Our Symphony further contributes to the preceding guidelines with a group-wise validation framework (Section III-A). It computes proofs with a group of packets as the input and then distributes proofs across the group.

**Initialization.** As aforementioned, we follow most existing solutions to assume a trusted source in practice [9], [10], [33], [36], [48]. Source $R_0$ prepares for proof construction by 1) exchanging shared symmetric key $K_i$ with each on-path router $R_i$ and 2) exchanging shared symmetric key $K_n$ with the destination $R_n$ (lines 1-3 in Algorithm 1). A common method used for the key exchange process is Diffie-Hellman [18], [23]. This enables the trusted source to pre-compute proofs used for verification for the routers and destination, greatly improving efficiency [33].

**Proof construction.** Through this function (lines 4-21 in Algorithm 1), the source instantiates all fields of the Symphony Header belonging to two categories. One consists of the leading fields that can be easily determined while aggregating packets and the other includes on SymphonyProof that requires certain pre-computation using the former category.

We start with instantiating fields in the first category (lines 5-12 in Algorithm 1). SessionID records the identifier of the current communication session.The subsequent GroupTag—together with SessionID—uniquely identifies a group of packets in the session. It is assigned with the index of groups. OrderTag and GroupSize are assigned with the index of a packet in the group and the number of packets in the group, respectively. Timestamp is set as the creation time of the first packet in the group. Of all the mentioned fields, SessionID, GroupTag, and GroupSize are identical in all the packets while OrderTag is different (lines 5-7). In contrast, Timestamp resides in only the first packet of the group (lines 8-9). Then we form group payload $GP$ by concatenating payloads of all packets in the group (line 10), hash $GP$ and assign it to DataHash (line 11), and evenly distribute DataHash across packets to amortize storage overhead (line 12).

SymphonyProof in the second category further consists of two types of fields—an update field $\sigma$ that helps computing proofs and a series of verification fields $V_i$ that are used to verify the computed proofs (lines 13-21 in Algorithm 1). $V_i$ is pre-computed by the source and stays fixed throughout the entire forwarding process while $\sigma$ is updated hop by hop:

$$\sigma \leftarrow \mathrm{MAC}_{K_n}(h), \tag{2}$$

where $K_n$ is the shared symmetric key between the source and destination and $h = \mathrm{SessionID}||\mathrm{Timestamp}||\mathrm{DataHash}$. The computation of $\sigma$ involves all of SessionID, Timestamp, and DataHash such that proof verification can simultaneously check data integrity and freshness. Similar to DataHash, $\sigma$ is also divided and distributed across all packets to amortize space overhead (line 17).

We iteratively compute a series of $V_i$ using $\sigma$ as follows (lines 18-21 in Algorithm 1). Since $R_i$ receives $\sigma_{i-1}$ computed by $R_{i-1}$, we can pre-compute verification field $V_i$ for $R_i$ with $\sigma_{i-1}$ as an input [33]. $R_i$ will accordingly verify $V_i$ carried in a received packet using the $\sigma_{i-1}$ carried therein (lines 22-31 in Algorithm 1). Since $R_{i-1}$ is the only router that is supposed to correctly compute $\sigma_{i-1}$, a successful verification of $V_i$ using the carried $\sigma_{i-1}$ demonstrates that the packet has actually traversed $R_{i-1}$. Similarly, $R_i$ embeds its credential by updating $\sigma_{i-1}$ to $\sigma_i$ (lines 32-34 in Algorithm 1), which is then used for $R_{i+1}$ to verify $V_{i+1}$.

$$\sigma_i = \mathrm{MAC}_{K_i}(\sigma_{i-1}). \tag{3}$$

$$V_i = \mathrm{MAC}_{K_i}(h||\sigma_{i-1}). \tag{4}$$

We then divide and distribute $V_i$ among the group.

**Proof verification.** Once after $R_i$ receives a complete ordering group of packets correlated by their SessionID, GroupTag, OrderTag, and GroupSize, it verifies the validity of $V_i$ (lines 22-31 in Algorithm 1). $R_i$ first computes $V_i'$ in the same way as the $V_i$ should have followed:

$$V_i' = \mathrm{MAC}_{K_i}(h||\sigma). \tag{5}$$

If $V_i'$ matches $V_i$ carried in the group of packets, verification succeeds. Otherwise, verification fails. Another case that fails verification is when $R_i$ finds an incomplete group due to packet losses or a long delay. In Algorithm 1, we handle verification failures with an all-or-nothing principle for ease of design. Specifically, if a group of packets fails verification, the entire group is discarded.

**Proof update.** If $R_i$ successfully verifies $V_i$ and is not the destination, it needs to update $\sigma$ for integrating its own credential (lines 32-34 in Algorithm 1).

$$\sigma = \mathrm{MAC}_{K_i}(\sigma). \tag{6}$$

Then $R_i$ divides the updated $\sigma$ into all packet headers and forwards the verified group of packets to the next hop $R_{i+1}$. The updated $\sigma$ therein is used by $R_{i+1}$ for `Verification`.

## V. Security

In this section, we prove that Symphony satisfies both security requirements—proof unforgeability and hop-wise validation—defined in Section II-C. We also discuss its resistance to distributed denial-of-service (DDoS) attacks.

### A. Proof Unforgeability

Proof unforgeability is the major security goal of any path validation solution. It requires that the attacker can hardly forge a valid proof. We use Cipher Block Chaining MAC (CBC MAC) [1] as the validation primitive. Therefore, proof unforgeability depends on the security of CBC MAC. A secure MAC must defend the adaptive chosen-message attack to resist potential forgery [4], [12], [21]. As for CBC MAC, its security is bounded by the insecurity of a CBC$^m$-$F$, where $F$ is a

pseudo-random function (PRF) or pseudo-random permutation (PRP), as shown in Theorem 1 [3].

**Theorem 1.** *[3] Let $l, m \geq 1$ and $q, t \geq 1$ be integers such that $qm \leq 2^{(l+1)/2}$. Let F: Keys(F)$\times\{0,1\}^l \rightarrow \{0,1\}^l$ be a family of functions. Then*

$$\mathbf{Adv}^{mac}_{CBC^m\text{-}F}(q,t) \leq \mathbf{Adv}^{prf}_{F}(q',t') + \frac{3q^2m^2+2}{2^{l+1}} \leq \mathbf{Adv}^{prp}_{F}(q',t') + \frac{2q^2m^2+1}{2^{l}},$$

*where q'=mq and t'=t+O(mql).*

Theorem 1 shows that the probability of breaking CBC MAC is no more than that of directly breaking the function $F$ used in CBC MAC with comparable resources. $\mathbf{Adv}^{\text{mac}}_{\text{CBC}^m\text{-}F}(q,t)$ denotes that among all the attackers whose restricted resources are $q$ messages and $t$ execution time, the maximum advantage the attacker has when distinguishing a random example of CBC$^m$-$F$ from a PRF. Similarly, $\mathbf{Adv}^{\text{prf}}_{F}(q',t')$ and $\mathbf{Adv}^{\text{prp}}_{F}(q',t')$ denote the maximum advantage the attacker has when distinguishing $F$ from a PRF or a PRP with restricted resources, respectively. In this way, we can cryptanalyze the lower-level primitive $F$ to estimate the security level of the original CBC MAC. Symphony uses AES-128 as the underlying block cipher of CBC MAC. Suppose that there exists a practical method that has a probability of forging messages by executing an adaptive chosen-message attack after accessing a large number of MACs of messages, say $2^{20}$ 1,000-byte packets. Then according to Theorem 1, there should be a practical method that has a similar probability of differentiating AES values at $2^{26}$ points among many random distinct points with the same restricted resources, associating with a very low probability of $2^{-26}$.

Furthermore, Symphony guarantees proof unforgeability against brute-force attacks in the following three ways.

First, the probability of forging a Symphony proof via brute-force attacks is negligible. Our Symphony proof cryptographically computes auxiliary fields (i.e., SessionID, Timestamp, and DataHash) into two validation fields (i.e., $\sigma$ and $V_i$). According to Symphony configuration, the probability of forging $\sigma$ and $V_i$'s for a group of packets along an $n$-hop path is $2^{-128(n+1)}$.

Second, the attacker can hardly crack the shared keys between the source and other nodes. If the attacker could crack all such shared keys, it need not forge valid proofs via brute-force attacks. Instead, it can freely forge any proof as if it were the trusted source. Symphony uses a 128-bit security level that is sufficiently secure for symmetric encryption.

Third, the attacker has to forge proofs over an entire group instead of a single packet as in existing solutions. This further hinders the attacker from forging proofs. Specifically, Symphony divides proofs into packets in a group. A correct proof cannot be computed using only a single packet as the input. Group-wise forging complicates the attacker by an $\mathcal{O}(m)$ complexity, where $m$ denotes the group size.

### B. Hop-wise Validation

We further prove that Symphony is robust against attacks such as packet alteration (Theorem 2), injection (Theorem 3), and deviation (Theorem 4).

**Theorem 2.** *Once any packet of a group is modified, the alteration can be detected and the group of packets cannot pass validation.*

*Proof:* Suppose the group $G$ sent by source $R_0$ is altered to $G'$ prior to its arrival at $R_i$.

$\forall(G, R_0, s), (G', R_i, r)$s.t.: $G, G' \neq \perp \wedge 1 \leq i \leq n \wedge R_i \in \Phi$
If $G.GP \neq G'.GP$
$\Rightarrow G.DataHash \neq G'.DataHash$
$\Rightarrow \Psi(G', R_i) = 0$

Given that the attacker can hardly forge proof fields in packet headers, it may directly modify packet payloads. Any modification of packet payloads necessitates recomputing the corresponding DataHash to pass integrity check. Group $G'$ thus carries a new DataHash different from the original one carried by $G$. However, the source signs DataHash into proofs using its shared key with the destination (lines 14-15 in Algorithm 1). Thus, the validation in $R_i$ would fail. ■

**Theorem 3.** *Once some packets are injected, routers can detect the attack and prevent injected packets from passing validation.*

*Proof:* Suppose that the group $G_a$ arriving at $R_i$ includes some injected packet(s).

$\forall(G_a, R_i, r)$ s.t.: $G_a \neq \perp \wedge R_i \in \Phi$
If $G_a$ is not entirely generated by source
$\Rightarrow G_a.proof$ is not signed by source
$\Rightarrow G_a.V_1, ..., G_a.V_i$ are not valid
$\Rightarrow \Psi(G_a, R_i) = 0$

Let $G_b$ denote any group sent by source $R_0$.

$\forall(G_a, R_i, r), (G_b, R_j, s)$ s.t.: $G_a, G_b \neq \perp \wedge R_i, R_j \in \Phi$
If $G_b$ is replayed by $G_a$
$\Rightarrow G_a.Timestamp = G_b.Timestamp$
$\Rightarrow G_a$ is expired
$\Rightarrow \Psi(G_a, R_i) = 0$

The first set of equations shows how we prevent injected packets from passing validation. If the group is generated by intermediate routers, the proof fields carried in the header are not signed by the source using corresponding symmetric keys and are not valid. Since attackers can hardly forge valid proofs for the injected packets to pass verification, benign routers can detect injected packets and drop them.

Another potential alternative is a replay attack, in which the attacker directly replays some valid packet(s) it captured. If a replayed packet is injected to a certain group, it is regarded as a forged packet to the group and cannot pass validation as the second set of equations shows. If the attacker captures and relays an entire group of valid packets, they can be detected and filtered using Timestamp [33]. Once a router detects an already existed Timestamp, it considers the group as expired and drops the group. Since Timestamp is also signed into unforgeable proofs (lines 14-15 in Algorithm 1), the attacker cannot evade detection by forging Timestamp in the replayed packets either. ■

**Theorem 4.** *Once the packets are deviated, the mis-forwarding can be detected and the group cannot pass validation.*

*Proof:* In the first case, suppose the group $G_a$ sent by $R_i$ is deviated to an off-path router $R_j$.

$\forall (G_a, R_i, s), (G_b, R_j, r)$ s.t.: $G_a, G_b \neq \perp \wedge i \neq j \wedge R_i \in \Phi$
If $R_j \notin \Phi$
$\Rightarrow G_b.SessionID$ cannot match with $\Phi$
$\Rightarrow \Psi(G_b, R_j) = 0$

In the second case, suppose that the packets are forwarded in an incorrect order like $\{R_i, R_k, R_j\}$ (while the correct order is $\{R_i, R_j, R_k\}$).

$\forall (G_a, R_i, s), (G_b, R_k, r), (G_c, R_k, s), (G_d, R_j, r)$
s.t.: $G_a, G_b, G_c \neq \perp \wedge R_i, R_k, R_j \in \Phi$
If $i + 2 = k + 1 = j$
$\Rightarrow V_k \neq \mathrm{MAC}_{K_k}(h||\sigma_i)$
$\Rightarrow \Psi(G_b, R_k) = 0$

The misbehavior of deviation is against Requirement 1. Symphony can detect it and stop the group from passing verification. Specifically, the attacker launches such attacks through a compromised router. It may deviate packets away from the specified forwarding path in two fashions. First, it forwards packets to a different path from the one bounded to SessionID. Second, it forwards packets to some router on the specified path by sidestepping the specified next-hop router. The above two sets of equations correspond to these two cases. In the first case, the path that the packet is mis-forwarded to should maintain a different SessionID than that carried in the packet, because Symphony enforces a SessionID-path binding. The router receiving the mis-forwarded packet can easily detect the mismatching of SessionIDs and discard the group including mis-forwarded packets. In the second case, the sidestepped router will not integrate its credential into the path proofs (line 34 in Algorithm 1). This results in the router not being able to calculate a correct and valid proof. This fails proof verification on the router that receives the mis-forwarded packet (lines 25 in Algorithm 1). ■

### C. *DDoS Resistance*

Finally, we discuss Symphony's resistance to DDoS attacks. Such attacks targeting our Symphony in particular might exploit two vectors—packet alteration and packet loss.

**Packet alteration.** As with existing path validation solutions, tampering with any proof-computation–related field (e.g., Timestamp) prohibits a packet from being validated. Packets failing validation will be discarded and retransmitted. This resembles packet alteration attacks (e.g., integrity violation) in traditional networks without path validation. A DDoS attack occurs if proof-field alteration affects a large number of packets. However, it is not among the initial design goals of path validation to defend against such DDoS attacks (Section II-C). Proofs computed for path validation aim more at unforgeability such that a packet cannot claim its forwarding along a path it has not actually taken. To augment path validation with DDoS resistance, a possible choice is first to locate routers on which DDoS attacks originate and then to exclude them from path choices. We hereby suggest a feasible method to locate suspicious routers. Specifically, each router tracks the number of invalid packets from every previous-hop router. Routers sending an abnormally high number of invalid packets can be considered suspicious for launching DDoS attacks.

**Packet loss.** In comparison with existing packet-wise path validation, our Symphony is more sensitive to packet losses. A lost packet induces retransmission of only one packet in packet-wise path validation. In aggregate validation featured by Symphony, however, a lost packet fails the verification of an entire group of packets and necessitates retransmitting all of them. This seems to render Symphony more vulnerable to DDoS attacks that exploit packet losses. To investigate this concern, we evaluate the impact of packet losses on throughput in Section VI-J. Both solutions do experience throughput reduction upon packet losses. It turns out that Symphony can still yield higher throughput than state-of-the-art EPIC does when the packet loss rate is as practically high as 10%. Against potentially higher packet loss rates under disruptive network conditions, we explore a packet reaggregation technique in Section VI-K. Once a group experiences lost or unverified packets, packet reaggregation enables an intermediate router to function as a new source, re-initiate aggregation validation over the remaining verified packets of the group, and only retransmit lost or unverified packets as in traditional packet-wise validation solutions. Furthermore, we do not send lost or unverified packets individually. We mix them into subsequent packet groups to regain efficiency from aggregate validation.

## VI. Evaluation

We implement Symphony using DPDK [28] and practice it on a local server as well as on a real deployed multihop testbed for performance evaluation. First, the local experiments are carried out on an Intel Xeon E5-2630 v3 (2.40 GHz) server [29]. It has 8 cores, 16 GB memory, and two hardware network-interface cards (NICs). The NICs are associated with Intel Corporation I350 Gigabit Network Connection and Intel Corporation Ethernet Controller X710. To measure the throughput, one NIC serves as a packet generator and bandwidth monitor, and the other performs computation of path validation. Second, the multihop testbed adopts rented c6e servers (with 3.2 GHz Intel Xeon Platinum 8269CY processors) from cloud computing services by Alibaba Cloud [17]. Both settings support a 10 Gbps link. We use the Intel AES-NI hardware instruction [41] to further accelerate the cryptographic computation. In comparison with state-of-the-art EPIC [36], Symphony speeds up packet processing by $3.78\times\sim18.40\times$ and increases communication throughput by $1.13\times\sim6.11\times$.

### A. *Proof Size*

We start with evaluating the communication overhead by Symphony proofs. The metric is the proof size. We expect the proofs to be as short as possible such that much bandwidth is left for transmitting payloads. The length of the entire proof is dominated by the path length. Given a fixed path length, a larger group imposes a shorter proof on each packet on average. Shorter proofs, however, do not necessarily yield better performance as a larger group may enforce a longer delay. In practice, we consider it possible to use an adaptive strategy to adjust group size in response to real-time network conditions. We adhere to common configurations in related work in what follows.

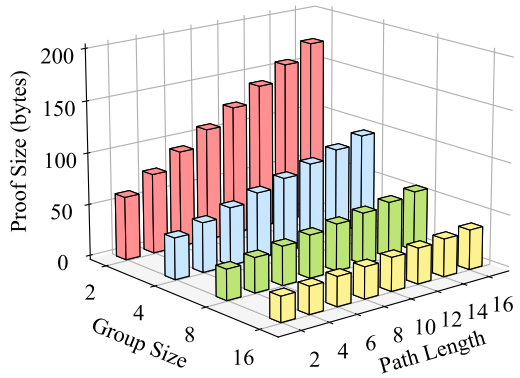



Fig. 3. Proof size comparison of Symphony under a 128-bit security level with varying path lengths and group sizes.

**Symphony proof size.** As shown in Table IV, the size of a Symphony header under a fixed security level depends on two parameters—path length $n$ and group size $m$. The payload size does not affect proof size because we first compute the hash result (i.e., DataHash) of the payload and then use a constant-size DataHash for proof computation. Following the structure of a Symphony header in Figure 2, the size of the first packet in a group can be defined as:

$$23 + \frac{16(n+3)}{m}.$$

The only difference between the first packet and the other $m-1$ packets in the same group is that the latter ones do not contain Timestamp in their headers. We estimate the average proof size per packet as follows.

$$\frac{23 + \frac{16(n+3)}{m} + (19 + \frac{16(n+3)}{m})(m-1)}{m} = \frac{19m + 16n + 52}{m}.$$

TABLE IV. SIZE AND QUANTITY OF EACH FIELD IN A SYMPHONY HEADER (ALGORITHM 1) WITH AN $m$-PACKET GROUP, AN $n$-HOP PATH, AND A 128-BIT SECURITY LEVEL. (NOTE THAT '*' REPRESENTS THAT THE FIELD IS ENTIRELY STORED IN THE FIRST PACKET OF A GROUP.)

| Field | Size (byte) | Quantity |
|---|---|---|
| SessionID | 16 | 1 |
| GroupTag | 1 | 1 |
| OrderTag | 1 | 1 |
| GroupSize | 1 | 1 |
| Timestamp* | 4 | 1 |
| DataHash | $32/m$ | 1 |
| $\sigma$ | $16/m$ | 1 |
| $V_i$ | $16/m$ | $n$ |

Figure 3 illustrates the proof size of Symphony under a 128-bit security level. Since each part of $V_i$ takes $16/m$ bytes in a packet header, we configure it as an integer for ease of storage. Thus, we choose group size $m = 2, 4, 8, 16$ to measure the proof size. We have two observations over proof size variation. First, given a fixed path length, the proof size decreases with the group size because a larger group offers more packets to amortize the proof fields (e.g., $V_i$). For example, given an 8-hop path, the proof size of each packet decreases from 64 bytes to 41 bytes when the group size increases from 4 to 8. Second, given a fixed group size, the proof size increases with the path length because a longer path enforces more verification fields $V_i$ in each packet header. Using the smallest group size of 2 for example, Symphony yields a proof size of 109 and 173 bytes when the path length is 8 and 16, respectively.

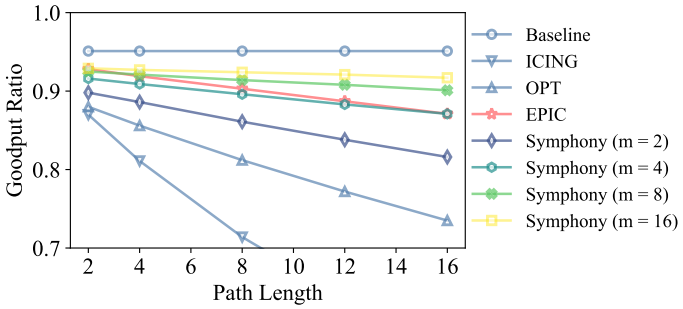



Fig. 4. Comparison of goodput ratio of Symphony with existing path validation solutions with 1,000-byte payloads and varying path lengths and group sizes.

**Comparison with existing solutions.** Table V compares the proof size of Symphony with that of existing path validation solutions. Existing solutions enforce an $\mathcal{O}(n)$ space complexity given an $n$-hop path. Since Symphony amortizes proofs across packets in a group, they thus yield a lower space complexity of $\mathcal{O}(n/m)$, where $m$ denotes group size. Symphony outperforms ICING and OPT in terms of shorter proof sizes. However, Symphony may yield longer proofs than state-of-the-art EPIC does under certain settings. EPIC achieves such short proofs by embedding partial proofs in each packet. It requires additional security schemes to mitigate the so caused vulnerabilities [36].

### B. Goodput Ratio

We further quantify the impact of proof size on performance via goodput ratio. The goodput ratio represents the ratio of payload size to total packet size (including both packet header and payload) [36]. For baseline solutions without path validation incorporated, a packet header simply consists of an IP header and a TCP header, which are 20 bytes and 32 bytes, respectively. For path validation solutions, they introduce additional validation header fields with various sizes.

Figure 4 compares the goodput ratio of Symphony with that of existing solutions. Without loss of generality, we adopt an average payload of 1,000 bytes. The baseline solution delivers a constant goodput ratio of 95.06% regardless of path length. This is straightforward because it introduces no additional header fields for path validation and thus has the highest goodput ratio. For all the path validation solutions, the goodput ratio decreases with path length. Below the highest baseline is Symphony with GroupSize of 16. When the path length is 16, Symphony still has a goodput ratio of 91.72%. EPIC follows Symphony with GroupSize of 16. Again, EPIC yields such a high goodput ratio because it only embeds partial proofs in packet headers. Among the path validation solutions that embed complete proofs in packet headers, our Symphony outperforms OPT and ICING.

Note that proof size alone cannot comprehensively quantify validation performance. It also matters about how fast packets can be processed. For example, if a validation induces relatively long proofs yet offers fast packet processing, it may still quickly validate a volume of packets. We thus proceed with evaluating packet processing overhead at the source (Section VI-C) and router (Section VI-F).

### C. Construction Time at Source

Figure 5 compares the proof construction time per packet of Symphony and state-of-the-art EPIC [36]. We report the

TABLE V. PROOF SIZE COMPARISON OF SYMPHONY WITH EXISTING SOLUTIONS UNDER A 128-BIT SECURITY LEVEL WITH VARYING PATH LENGTH $n$ AND GROUP SIZE $m$.

| Solution | Proof Size | Group Size $m$, Path Length $n$ | | | | | |
|---|---|---|---|---|---|---|---|
| | | $m=4, n=8$ | $m=4, n=12$ | $m=8, n=8$ | $m=8, n=12$ | $m=16, n=8$ | $m=16, n=12$ |
| ICING [39] | $42n+13$ | 349 | 517 | 349 | 517 | 349 | 517 |
| OPT [33] | $16n+52$ | 180 | 244 | 180 | 244 | 180 | 244 |
| EPIC [36] | $5n+16$ | 56 | 76 | 56 | 76 | 56 | 76 |
| Symphony | $(19m+16n+52)/m$ | 64 | 80 | 41 | 49 | 30 | 34 |

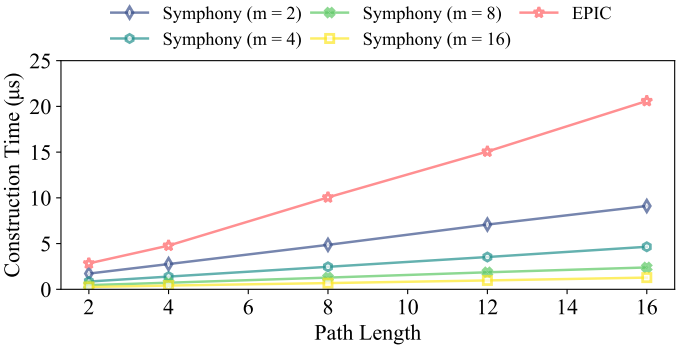



Fig. 5. Construction time per packet at the source with 1,000-byte payloads under varying path lengths and group sizes.

average construction time of 10,000 packets each with 1,000-byte payloads. The construction time of all solutions increases with path length as the source needs to compute an individual verification field for each hop. The construction time of Symphony with a 4-packet group increases from 0.85 $\mu$s to 3.53 $\mu$s when the path length increases from 2 to 12. For Symphony, the construction time decreases with group size because of the amortization effect across packets in a group. Consider, for example, when the path length is 16. Symphony takes 9.11 $\mu$s to construct proofs for a 2-packet group while only 2.39 $\mu$s for a 8-packet group.

Symphony outperforms EPIC. The average reduction of construction time ranges from 39.51% (Symphony with group size of 2) to 93.78% (Symphony with group size of 16).

### D. Reassembly Time on Router

Prior to evaluating aggregate validation time in Section VI-F, we first evaluate the time for aggregating a group of packets in Section VI-D and for reordering them in Section VI-E.

TABLE VI. GROUP REASSEMBLY TIME WITH VARYING GROUP SIZES.

| Reassembly Time | Group Size $m$ | | | |
|---|---|---|---|---|
| | $m=2$ | $m=4$ | $m=8$ | $m=16$ |
| Least Upper Bound ($\mu$s) | 0.06 | 0.13 | 0.17 | 0.29 |

We use a simple yet effective time-bounded group reassembly method. The key idea is to first discover the least upper bound of time that a router takes to receive a complete group of packets and then use this time bound as a threshold. When the waiting time exceeds the threshold and the router cannot find all expected groups belonging to a certain group, the router ascertains potential packet losses and considers the group as incomplete. The time bound can be collaboratively discovered by an end-host and the first-hop router. The end-host keeps sending continues groups of test packets to the first-hop router. Then the router simply tracks the time for receiving each group. Given relatively stable network conditions, reassembly time per group will quickly converge and reveal the feasible time bound. We report group reassmebly time with varying group sizes in Table VI. It is much faster to reassemble a group than to validate it (Section VI-F).

### E. Reordering Time on Router

We investigate the time for reordering a group of received packets in two cases—hardware router and software router.

**Hardware router.** We have suggested an SRAM-based hardware router logic to implement packet reordering in Section III-B. The order of packets in a group is correlated to cell locations on the SRAM. Once a router finishes fetching a group of packets from the queue and placing them in corresponding cells, it already obtains a group of ordered packets. No further reordering effort is needed. In this case, reordering timer per packet approximates to the SRAM access time (e.g., 0.003 $\mu$s [44]).

TABLE VII. PACKET REORDERING TIME WITH VARYING GROUP SIZES.

| Reordering Time | Group Size $m$ | | | |
|---|---|---|---|---|
| | $m=2$ | $m=4$ | $m=8$ | $m=16$ |
| Average Time ($\mu$s) | 0.01 | 0.01 | 0.02 | 0.03 |
| Upper Bound ($\mu$s) | 0.01 | 0.03 | 0.04 | 0.06 |

**Software router.** On the software router—DPDK—we use, sorting is necessary for reordering a group of packets. We choose bubble sort [2] because it offers fast and efficient sorting for relatively small groups. In contrast, other alternatives (e.g., quicksort) may require additional parameters (e.g., pivot) that consume packet space and router resource. Table VII shows the average reordering time and the upper bound per packet. When the group size increases from 2 to 16, the average packet reordering time increases from 0.01 $\mu$s$\sim$0.03 $\mu$s. The corresponding ratio of reordering time to verification time (Section VI-F) ranges from 1.57% to 20.68%.

### F. Processing Time on Router

Figure 6 reports the average processing time per packet on each router of Symphony in comparison with that of EPIC. It consists of the time for verifying proofs and for updating proofs. We report the average processing time over 10,000 packets with varying path length and group size. The path length does not quite fluctuate the processing time. A router only needs to perform two MAC operations to verify and update proofs. The related fields $\sigma$ and $h$ do not depend on the path length. For example, the processing time of Symphony with a 16-packet group keeps around 0.13 $\mu$s$\sim$0.15 $\mu$s as the path length increases. The tiny increase is brought by

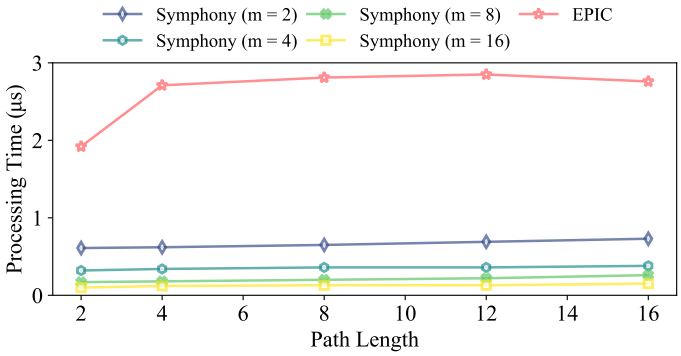
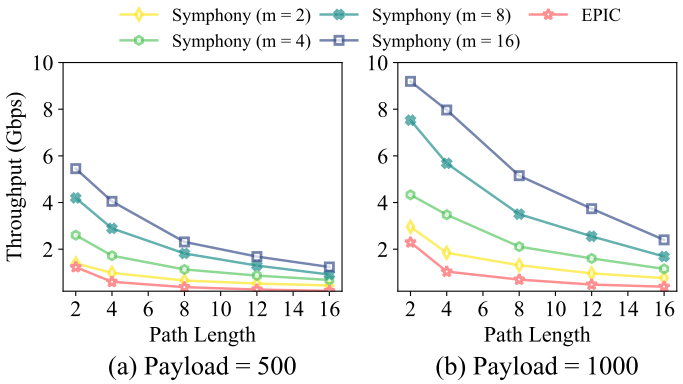



Fig. 6. Processing time per packet on the router with 1,000-byte payloads at the source with varying path length and group size.




(a) Payload = 500 (b) Payload = 1000

Fig. 7. Comparison of throughput with varying payload sizes, path lengths, and group sizes.

splitting and merging related proof fields. In contrast to the path length, the group size matters for the processing time of Symphony. Since our solutions leverage group-wise validation, a larger group offers more packets to amortize the processing time. For example, when the path length is 12, the processing time of Symphony is 0.69 $\mu$s given a group of 2 packets. When GroupSize reaches 8, the processing time shrinks to only 0.22 $\mu$s, being 68.12% faster. The second conclusion is that despite processing time and group size are positively correlated, they do not have a strict multiplicative relationship. Considering a 16-hop path, the processing time decreases from 0.73 $\mu$s to 0.15 $\mu$s as the group size increases from 2 to 16. The speed only increases by 4.87 times, which is significantly lower than the actual increase in group size. This is because the merging and splitting operations become more complicated when the group size is larger while the original verification operation keeps the same.

Symphony yields a much shorter processing time than EPIC does. As shown in Figure 6, even the bottom case of Symphony (0.73 $\mu$s) with GroupSize = 2 is about 3.78× faster than EPIC (2.76 $\mu$s) when the path length is 16 and the payload size is 1,000 bytes. In the same settings, Symphony with GroupSize = 16 further reduces the processing time to 0.15 $\mu$s, being 18.40× faster than EPIC.

### G. Throughput and Goodput

**Throughput.** Figure 7 reports the throughput measurements of Symphony and EPIC. The source limits throughput scale because it takes more time to process packets than other routers on the same path (Figure 5 and Figure 6). To concentrate on efficiency by validation primitives per se, we run the experiments with a single CPU without leveraging multi-CPU parallelism (to be evaluated in Section VI-I). Symphony achieves higher throughput than EPIC does.

We now discuss how throughput varies with payload sizes, path lengths, and group sizes. First, throughput increases with payload sizes. Since the payload size does not affect the processing time much, a larger payload size yields more data to transmit in a given time. For example, when the payload size increases from 500 bytes to 1,000 bytes, throughput of Symphony with group size of 4 increases from 2.60 Gbps to 4.33 Gbps on a 2-hop path. Second, throughput decreases with path length. As the construction time increases with the path length, a fixed packet size results in the decrease of throughput. When the path length increases from 2 to 12, the throughput of Symphony decreases by 69.07% with an 8-packet group. Third, when GroupSize increases, the average processing time of each packet decreases while the total packet size increases. Thus, throughput increases with GroupSize. When the path length is 2 and the payload size is 1,000 bytes, the throughput of Symphony is 9.19 Gbps with group size of 16, increasing by 112.26% in comparison with when the group size is 4. Fourth, the increase in throughput is less than the increase in GroupSize. When a group becomes larger, merging and splitting proofs become more complicated, resulting in a slower increase of throughput. For example, when the group size increases by 8× from 2 to 16, the throughput only increases by 3.15× with a 16-hop path and a 1,000-byte payload. This indicates an important conclusion that the increase in group size has a threshold, especially considering situatiosn with packet loss and retransmission (Section VI-J). In summary, Symphony yields a higher throughput ranging from 1.13× that of EPIC given payload size of 500, path length of 2, and group size of 2 to 6.11× that of EPIC given payload size of 1,000, path length of 16, and group size of 16.

**Goodput.** Goodput ultimately reflects how fast data transmission can be. The goodput ratio is the ratio of payload size to total packet size (Section VI-A). It can also be equivalently defined as the ratio of goodput to throughput [36]. Therefore, we can estimate the goodput as:

$$\text{goodput} = \text{throughput} \times \text{goodputratio}. \tag{7}$$

We measure the goodput based on the throughput in Figure 7 and compare our solutions with EPIC in Figure 8. Symphony achieves a higher goodput than EPIC does, increasing by 1.09× (with payload size of 500, path length of 2, and group size of 2) to 6.56× (with payload size of 1,000, path length of 16, and group size of 16).

### H. Sensitivity to Mixed Packet Size

Figure 9 reports performance measurements with mixed packets of different sizes in a group. Given that processing time depends more on packet header fields rather than payloads, composition of intra-group packets does not fluctuate processing time. Therefore, varying packet sizes affects only the scale of throughput and goodput. It does not necessarily affect the performance comparison effect of various solutions given the same settings. We provide in Figure 9 with an instance of mixed-size packets emulating video traffic—1/3 of the group are 480-byte packets for audio transmission and the

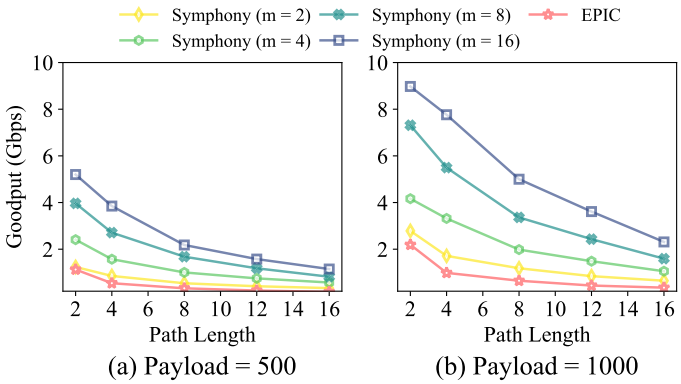



Fig. 8. Comparison of goodput with varying payload sizes, path lengths, and group sizes.

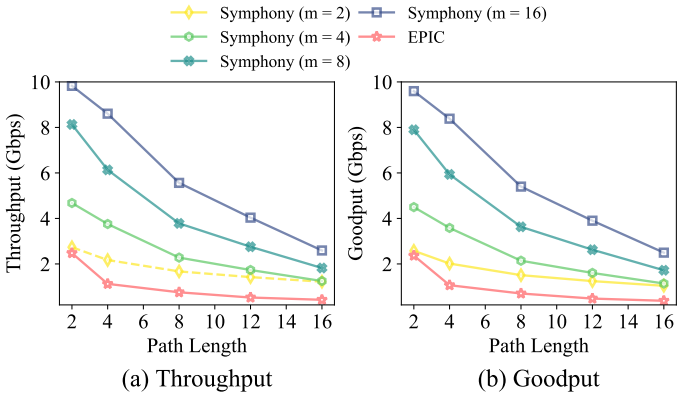



Fig. 9. Comparison of throughput and goodput with varying path length and a 4-packet group of mixed-size packets emulating video traffic.

remaining 2/3 of the group are 1,400-byte packets for video transmission [35]. The performance trend of Symphony with different group sizes is basically the same as that shown in Figure 7 and Figure 8. Various other mix-size packet instances exhibit similar trends.

### I. *Multi-core Acceleration*

We evaluate the performance of multi-core acceleration for Symphony. Due to the limit of 16 GB server memory, the maximum cores used are limited to 4 as a larger setting would lead to memory outage. Figure 10 reports Symphony throughput with varying core count, payload size, and group size on a 4-hop path. The throughput increases with payload size and group size as we discussed in Section VI-G. We observe that throughput increases with allocated cores. With a 1,000-byte payloads and an 8-packet group, Symphony achieves a 5.68 Gbps throughput using only one core. It is accelerated to 8.06 Gbps when given up to 2 cores. The throughput further reaches the upper bound of 10 Gbps link while using 3 and 4 cores.

### J. *Packet Loss*

We evaluate the impact of packet loss rates on the throughput of Symphony. Considering that throughput quantifies the amount of data successfully transmitted per unit time, higher packet loss rates lead to lower throughput. Symphony needs to retransmit an entire group of packets as long as at least one packet in the group is lost. This leads to a potential source of network congestion. We use the TCP BBR model [13],

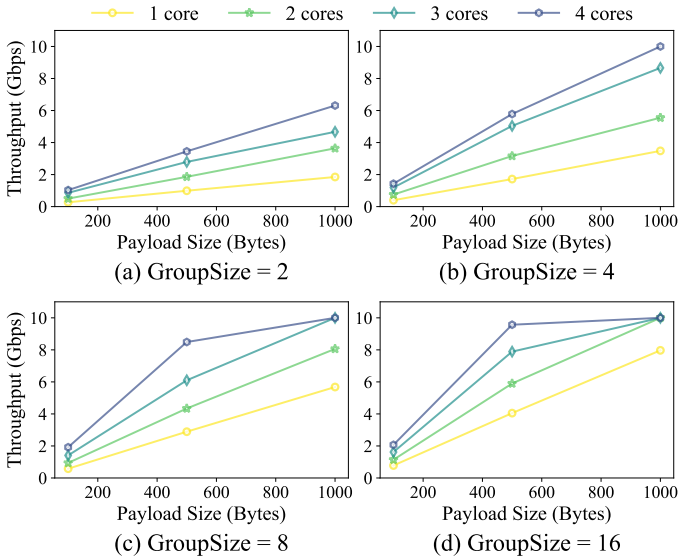



Fig. 10. Symphony throughput with varying core counts, payload sizes, and group sizes on a 4-hop path.

[14], [46]—a new congestion control algorithm—to estimate the impact. Assume that the packet loss rate of each packet is $p$. The entire group with $m$ packets in Symphony has a packet loss rate of $1-(1-p)^m$, while the successful delivery rate is $1-(1-(1-p)^m) = (1-p)^m$. The theoretical throughput of Symphony can be estimated as follows.

$$\text{Throughput} = T \times (1-p)^m, \tag{8}$$

where $T$ denotes the original throughput without packet losses. Table VIII shows the theoretical reduction in throughput for both EPIC and Symphony under various packet loss rates normalized over throughput without packet losses. EPIC retransmits only lost packets. In contrast, Symphony is more sensitive to packet losses due to retransmission of an entire group even upon a single packet loss. Larger group sizes thus lead to greater reduction in throughput.

TABLE VIII. THROUGHPUT REDUCTION WITH 1,000-BYTE PAYLOADS, PATH LENGTH OF 8, AND VARYING PACKET LOSS RATES.

| Solution | Packet Loss Rate | | | | | |
|---|---|---|---|---|---|---|
| | 0.1% | 0.5% | 1% | 3% | 5% | 10% |
| EPIC | 0.10% | 0.50% | 1.00% | 3.00% | 5.00% | 10.00% |
| Symphony ($m = 2$) | 0.20% | 1.00% | 1.99% | 5.91% | 9.75% | 19.00% |
| Symphony ($m = 8$) | 0.80% | 3.93% | 7.73% | 21.63% | 33.66% | 56.95% |

Figure 11 further reports the evaluation results of the throughput and related reduction for both solutions. Following the preceding analysis, packet losses affect Symphony with larger groups (e.g., 8-packet groups) more than Symphony with smaller groups (e.g., 2-packet groups) and EPIC without using packet groups. For example, given a packet loss rate of 1%—a threshold of user experience about Voice over Internet Protocol (VoIP) [16]—the throughput reduction of EPIC, Symphony with 2-packet groups, and Symphony with 8-packet groups is 0.01 Gbps, 0.07 Gbps, and 0.38 Gbps, respectively. When the packet loss rate increases to 5%, throughput of EPIC, Symphony with 2-packet groups, and Symphony with 8-packet groups further decreases by 5.01%, 11.59%, and 33.75%, respectively, in line with theoretical results in Table VIII. Albeit more sensitive to packet losses, Symphony still outperforms

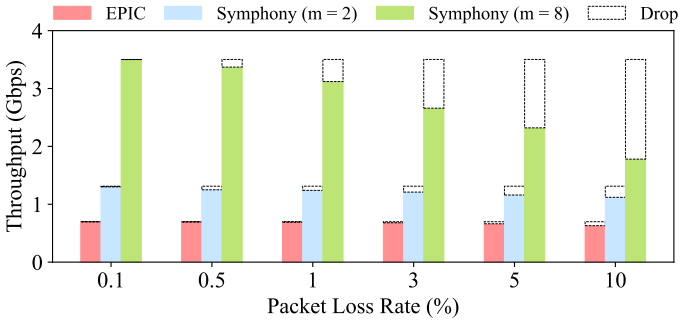



Fig. 11. Comparison of throughput with varying packet loss rates on an 8-hop path with 1,000-byte payloads.

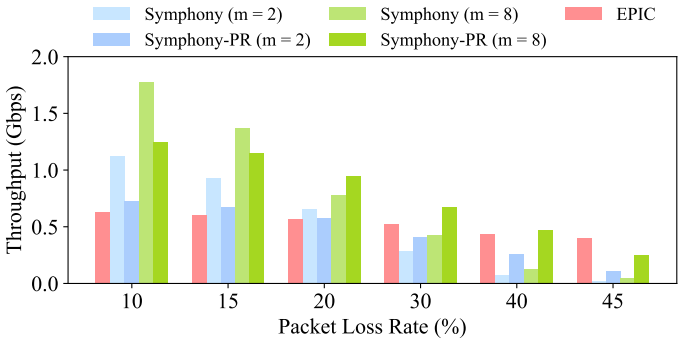



Fig. 12. Comparison of throughput with relatively high packet loss rates on an 8-hop path with 1,000-byte payloads.

EPIC with a higher throughput. This again demonstrates the effect of our proposed aggregate validation.

### K. Packet Reaggregation

We further explore a packet reaggregation technique to deal with even higher packet loss rates under disruptive network conditions. It enables an intermediate router to function as a new source and re-initiate aggregation validation over the remaining verified packets of a group. This way, only lost packets require retransmission.

**Verification of incomplete groups.** The first challenge for packet reaggregation is to identify valid packets in an incomplete group. Packet reaggregation is invoked when a group of packets fails verification. Two such cases are when 1) the group contains fewer packets than GroupSize claims and 2) the group satisfies GroupSize while some packets therein are invalid. The generic root cause for both cases is that the group is incomplete in terms of lacking valid packets. When we regroup the remaining packets, we need to make sure that they conform to path validation so far. However, the verification fields they carry should have been computed using an entire group as the input. We need to modify `Construction` in Algorithm 1 to support verifying packets in an incomplete group.

We leverage the unforgeability of verification fields $V_i$ to solve the challenge. Recomputing $V_i$ requires $\sigma$ (Equation 5 in Section IV-C). Taking into account packet losses, we no longer divide $\sigma$ into different packet headers. Otherwise, any missing packet leads to an incomplete $\sigma$ and thus fails recomputing $V_i$. Therefore, we choose to copy $\sigma$ in all packets during `Construction`. Upon reaggregation, we recompute $V_i$ using $h$ and $\sigma$ available in any packet[1]. Then we leverage the design fact that $V_i$ is distributed across packets during `Construction`. When $V_i$ is hard to forge, we expect that valid packets carry the corresponding segments of recomputed $V_i$. This helps a router to identify valid packets out of an incomplete group during reaggregation. The router then acts as a new source and performs aggregation validation over the new group of identified valid packets.

**Protection from untrusted new sources.** To enable a router as the new source, we require that end-hosts and routers share pair-wise secret key. It is critical to prevent the new source from manipulating its privilege. An untrusted new source may attack path validation in the following ways. First, an untrusted new source may reroute a packet to a different path. Then it substitutes the original SessionID with the one corresponding to the new path. The new SessionID is used for constructing the proofs. Second, an unstrusted new source may even modify packet payloads. Third, the new source is allowed to re-group packets and re-construct verification fields of the new group. Field $\sigma$ carrying credentials of nodes prior to the new source is supposed to be overwritten. An untrusted new source may turn a mis-forwarded packet into a valid one.

To combat the first two attacks, we introduce a signed binding between a packet and its forwarding path by the original source. Since the forwarding path of a packet can be vouched by SessionID, we construct the signed binding of packet $P$ as:

$$\delta = \text{MAC}_{K_{0n}}(\text{SessionID}||P.\text{payload}), \tag{9}$$

where $K_{0n}$ is the shared key between the original source and the destination. Any new source without the knowledge of $K_{0n}$ can hardly forge a valid $\delta$ after it reroutes a packet to a different path with a different SessionID or modifies the packet payload.

We further combat the third attack by introducing an unforgeable field $\epsilon$ to track the packet forwarding history. The original source constructs $\epsilon$ by nested encryption over the signed binding $\delta$ as:

$$\epsilon_i = \begin{cases} \text{Enc}_{K_{0n}}(\delta), & \text{if } i = 0; \\ \text{Enc}_{K_{0(n-i)}}(\epsilon_{i-1}), & \text{if } 1 \le i \le n-1. \end{cases} \tag{10}$$

Specifically, we iteratively encrypt $\delta$ using the shared keys with nodes backwards from the destination during `Construction`. Upon `Update`, $\epsilon$ is iteratively decrypted by one router after another. This associates $\epsilon$ with credentials of routers. If a packet traverses all the routers on the specified path in the correct order, $\epsilon$ received by the destination should be equal to $\delta$.

**Results.** We implement Symphony with Packet Reaggregation (i.e., Symphony-PR) and evaluate its resistance to disruptive packet losses. Figure 12 extends the results in Figure 11 with Symphony-PR for comparison under relatively high packet loss rates. In comparison with Symphony, Symphony-PR exhibits higher resistance to packet losses. It outperforms Symphony with 8-packet groups and Symphony with 2-packet groups when the packet loss rate reaches 20% and 30%, respectively. When the packet loss rate is as high as 45%, Symphony-PR with 2-packet groups and Symphony-PR with 8-packet groups yield a throughput 5.50 times and 5.00 times that of Symphony

[1]Note that $h$ and $\sigma$ should be identical in valid packets of the same group. Invalid packets with different $h$ and $\sigma$ can be easily detected and filtered. We hereby consider the attacker less incentive to inject invalid packets with incorrect $h$ and $\sigma$.

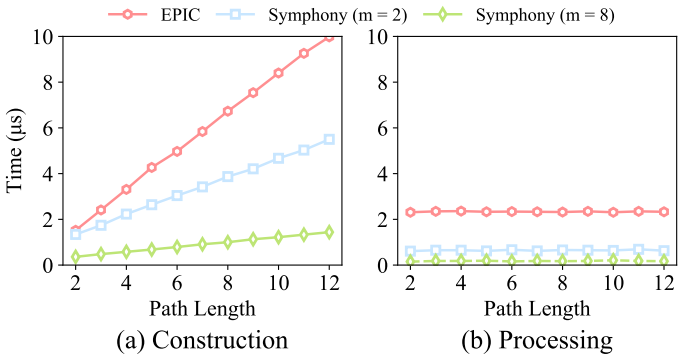



Fig. 13. Comparison of (a) construction time per packet on the source and (b) processing time per packet on the router with 1,000-byte payloads in a multihop testbed with varying path lengths and group sizes.

with 2-packet groups and Symphony with 8-packet groups, respectively. EPIC is less sensitive to packet losses than are both Symphony and Symphony-PR. As the packet loss rate increases from 10% to 45%, EPIC, Symphony with 2-packet groups, Symphony-PR with 2-packet groups, Symphony with 8-packet groups, and Symphony-PR with 8-packet groups reduce throughput by 36.51%, 98.21%, 84.93%, 97.19%, and 80.01%, respectively. Note that this does not necessarily render EPIC more efficient. Both Symphony and Symphony-PR can benefit from our proposed aggregate validation technique for efficiency. EPIC outperforms Symphony-PR with 8-packet groups only when the packet loss rate is disruptively high (e.g., 45%).

### *L. Multihop Testbed*

Finally, we conduct performance evaluation in a real deployed multihop testbed. The testbed is built on rented c6e servers from cloud computing services by Alibaba Cloud [17]. It supports up to a 12-hop path. Each c6e server features with four 3.2 GHz Intel Xeon Platinum 8269CY (Cascade Lake) processors. However, it allows only one 10 Gbps elastic NIC to be mounted and thus does not support multi-core acceleration as in Section VI-I. We hereby focus on single-core evaluation instead. Furthermore, given that Symphony keeps outperforming Symphony-PR and EPIC under normal network conditions with packet loss rates lower than 20% (Figure 12), we do not consider packet losses in testbed experiments.

Figure 13 reports the evaluation results of execution time. Both construction time on the source and processing time on each router get accelerated in comparison with the measurements on our local server (Figure 5 and Figure 6). This is because the rented server offers more computation resources. As with the comparison effect on our local server, Symphony continues to outperform EPIC. We first analyze the results of construction time in Figure 13(a). When the path length increases from 2 to 12, construction time of EPIC increases from 1.52 $\mu$s to 9.98 $\mu$s while that of Symphony with 2-packet groups and Symphony with 8-packet groups increases from 1.34 $\mu$s to 5.50 $\mu$s and from 0.36 $\mu$s to 1.44 $\mu$s, respectively. Symphony with 2-packet groups and Symphony with 8-packet groups thus respectively offer 1.13×∼1.81× and 4.22×∼6.93× over EPIC. We then discuss the results of processing time that take into account both proof verification and proof update in Figure 13(b). All solutions under evaluation provide a relatively constant processing time regardless of the path length. EPIC takes about 2.31 $\mu$s to verify a packet.

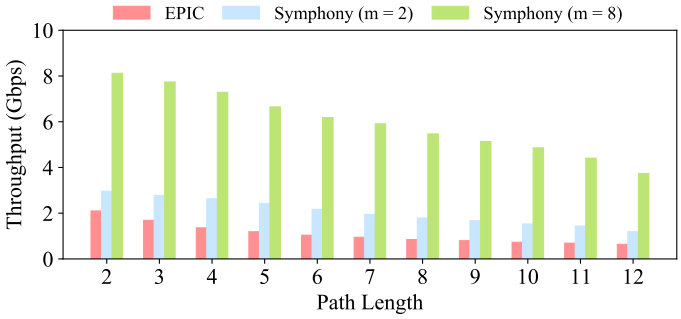



Fig. 14. Comparison of throughput with 1,000-byte payloads in a multihop testbed with varying path lengths and group sizes.

Symphony with 2-packet groups takes about 0.63 $\mu$s and is 3.67× faster than EPIC. Symphony with 8-packet groups further boosts processing speed and takes only about 0.17 $\mu$s, being 13.59× faster than EPIC.

Figure 14 shows the evaluation results of throughput. Since the source takes more time to construct proofs than a router takes to verify them (Figure 13), the source tends to be the bottleneck of throughput. Furthermore, longer paths cost the source more time to construct proofs as shown in Figure 13(a). Throughput thus decreases with path length (Figure 14). Given a fixed path length, both Symphony with 2-packet groups and Symphony with 8-packet groups yield higher throughput than EPIC does. Consider a 2-hop path for example. Throughput of EPIC, Symphony with 2-packet groups, and Symphony with 8-packet groups is 2.10 Gbps, 2.96 Gbps, and 8.12 Gbps, respectively. When the path length increases to 12, throughput of EPIC, Symphony with 2-packet groups, and Symphony with 8-packet groups becomes 0.64 Gbps, 1.19 Gbps, and 3.74 Gbps, respectively. In summary, Symphony with 2-packet groups and Symphony with 8-packet groups respectively achieve 1.41×∼1.86× and 3.87×∼5.84× higher throughput than EPIC does.

## VII. Related Work

In this section, we review existing path validation solutions. ICING [39] is acknowledged as the first path validation scheme with a highly strict security assumption, requiring pair-wise shared keys among end-hosts and routers. Subsequent solutions trade security for efficiency. For example, OPT [33] and most solutions afterwards improve computation efficiency by not enforcing pair-wise shared keys. They consider the source as trusted and require that symmetric keys be shared between only the source and every co-path node (i.e., each intermediate router and the destination). The source then pre-computes proofs for these co-path nodes using the shared keys. This helps to reduce the overall computation overhead and in turn improves efficiency. PPV [48] further adopts a probabilistic way to validate packets only at sampled routers instead of all routers along the path. OSV [9], [10] uses matrix computation to replace cryptographic computation. Such a substitution significantly improves the efficiency by simplifying the computation complexity. Finally, the state-of-the-art EPIC [36] carries only partial proof in packets to improve efficiency.

We next present the key design strategies of related work and compare them with Symphony proposed in this paper. Symphony identifies packet-wise validation as a fundamental efficiency bottleneck for existing solutions. It then explores

an aggregate validation technique to achieve efficient path validation in a group wise way.

### A. Evolution

ICING [39] requires every router to compute a proof for each downstream router separately and uses aggregate MAC [5], [31] to merge the original $\mathcal{O}(n^2)$ proofs into $\mathcal{O}(n)$ fields. Each field corresponds to an intermediate router on the path and is verified using upstream routers' proof. Once the verification succeeds, the router updates the proof and forwards the packet to the next hop. Each pair of nodes in ICING needs to exchange symmetric keys to compute the proof.

Origin and Path Trace (OPT) [33] decreases the number of proof fields that require verification and update into $\mathcal{O}(1)$. It has a PVF field and several OPV fields. Each OPV field is calculated by the source in advance based on PVF and related to a corresponding router. When a router receives a packet, it calculates a new proof using PVF and compares the result with the original one embedded in the header. If they match, verification succeeds and the router updates the PVF field and transfers the packet to the next hop.

PPV [48] uses a probability formula to select one on-path router to mark the packet rather than perform hop-wise validation. This router embeds proofs with its credential into the packet. Proofs are verified at the destination. Given a large amount of packets with each indicating the forwarding behavior of a path segment, the packets can jointly reflect the forwarding behavior of the entire path. Similarly, MASK [20] requires that the source select one intermediate router for a specific packet with the responsibility of proof generation. Once the packet reaches the selected router, the router calculates a proof with its credential. This proof is eventually verified by the destination. With sufficient packets, MASK can guarantee that all the on-path routers are covered and forwarding is secure.

OSV [9], [10] replaces traditional cryptographic computation like MAC with a Hadamard matrix. The matrix computation has a faster calculation speed and a better parallel computability. This makes OSV has a significant improvement on efficiency. However, its simplicity of computation results in that OSV cannot guarantee the security and integrity of packets during forwarding [10].

Improving upon OPT, state-of-the-art EPIC [36] shortens the length of the proof fields by only embedding the first three bytes of each OPV fields into packet headers. This significantly decreases the size of packet headers in comparison with existing schemes that store the entire proof. Like OPT, EPIC pre-computes an OPV field for each router. To accelerate the processing speed, proofs for intermediate routers are only a part of the original MAC values. Shortened proofs increase the risk of being forged.

### B. Comparison

In comparison with existing solutions, we consider our proposed aggregate validation a significantly new direction for path validation to breach the existing efficiency barrier. It functions more like a framework than a primitive. This is why the validation primitives we use can be from any existing scheme such as the state-of-the-art EPIC. In other words, implementing aggregate validation does not necessarily limit the choice of cryptographic primitives to symmetric cryptography. We currently use symmetric cryptography in line with most existing solutions because of its fast computation [9], [10], [33], [36], [39], [48]. Our protocol can also be built on asymmetric cryptography. As investigated in Atomos [24], asymmetric cryptography demands a relatively higher computation capability from routers and outperforms the symmetric-cryptography–counterpart given relatively long paths. The contribution of our proposal can be justified by its up to an order of magnitude performance improvement. New challenges are 1) investigating various design and implementation techniques for validating packets group wise, and 2) dealing with packet losses as even a single lost packet renders a group incomplete. We address these challenges with guaranteed security and efficiency.

## VIII. Conclusion

We have studied the idea of exploring aggregate validation for efficient path validation. It validates path compliance during packet forwarding in a group-wise way. In contrast to existing packet-wise validation, we aggregate a group of packets and use them together as a single input for computing path proofs. We then distribute proofs across packets in the same group and thus amortize validation overhead to improve efficiency yet without security being weakened. We implement aggregate validation through Symphony and adapt it to Symphony-PR toward more efficiently handling packet losses. The experiment results show that our solution yields faster packet processing and higher communication throughput than the state-of-the-art. For future work, we plan to extend Symphony with dynamic routing [25], [49] and privacy protection [25], [43]. We also plan to open source the Artifact.

## Acknowledgment

The work is supported in part by National Natural Science Foundation of China under Grant No. 62172358, National Key R&D Program of China under Grant No. 2020AAA0107700, and National Natural Science Foundation of China under Grant No. 62032021. We would like to sincerely thank NDSS 2024 Chairs, Shepherd, and Reviewers as well as Chairs and Reviewers of NDSS 2022, USENIX Security 2022, USENIX Security 2023, NDSS 2023, and CCS 2023 for your review efforts and helpful advice. All your thoughtful and constructive comments have guided us toward a much higher paper quality. We would also like to extend our gratitude to the authors that published and shared their source code to help us with performance evaluation.

## References

[1] ISO/IEC 9797. Data cryptographic techniques: Data integrity mechanism using a cryptographic check function employing a block cipher algorithm, 1989.

[2] Owen Astrachan. Bubble sort: an archaeological algorithmic analysis. *ACM Sigcse Bulletin*, 35(1):1–5, 2003.

[3] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences*, 61(3):362–399, 2000.

[4] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS*, pages 62–73, 1993.

[5] Andrea Bittau, Michael Hamburg, Mark Handley, David Mazieres, and Dan Boneh. The case for ubiquitous {Transport-Level} encryption. In *USENIX Security Symposium*, 2010.

[6] Bob Braden, David D Clark, Jon Crowcroft, Bruce S Davie, Steve Deering, Deborah Estrin, Sally Floyd, Van Jacobson, Greg Minshall, Craig Partridge, et al. Recommendations on queue management and congestion avoidance in the internet. *RFC*, 2309:1–17, 1998.

[7] Kai Bu, Avery Laird, Yutian Yang, Linfeng Cheng, Jiaqing Luo, Yingjiu Li, and Kui Ren. Unveiling the mystery of internet packet forwarding: A survey of network path validation. *ACM Computing Surveys*, 53(5):1–34, 2020.

[8] Kai Bu, Yutian Yang, Zixuan Guo, Yuanyuan Yang, Xing Li, and Shigeng Zhang. Flowcloak: Defeating middlebox-bypass attacks in software-defined networking. In *INFOCOM*, pages 396–404, 2018.

[9] Hao Cai and Tilman Wolf. Source authentication and path validation with orthogonal network capabilities. In *INFOCOM WKSHPS*, pages 111–112, 2015.

[10] Hao Cai and Tilman Wolf. Source authentication and path validation in networks using orthogonal sequences. In *IEEE ICCCN*, pages 1–10, 2016.

[11] Kenneth L Calvert, James Griffioen, and Leonid Poutievski. Separating routing and forwarding: A clean-slate network layer design. In *IEEE BROADNETS*, pages 261–270, 2007.

[12] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[13] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: congestion-based congestion control. *Communications of the ACM*, 60(2):58–66, 2017.

[14] Neal Cardwell, Yuchung Cheng, Soheil Hassas Yeganeh, and Van Jacobson. Bbr congestion control. *IETF Draft draft-cardwell-iccrg-bbr-congestion-control-00*, 2017.

[15] Laurent Chuat, Markus Legner, David A. Basin, David Hausheer, Samuel Hitz, Peter Müller, and Adrian Perrig. *The Complete Guide to SCION - From Design Principles to Formal Verification*. Information Security and Cryptography. Springer, 2022.

[16] Cisco. Video quality of service (qos) tutorial. https://www.cisco.com/c/en/us/support/docs/quality-of-service-qos/qos-video/212134-Video-Quality-of-Service-QOS-Tutorial.html, 2017.

[17] Alibaba Cloud. Compute-optimized instance families. https://www.alibabacloud.com/help/en/elastic-compute-service/latest/compute-optimized-instance-families#section-wiv-kqq-u7o.

[18] Whitfield Diffie and Martin E Hellman. New directions in cryptography. In *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*, pages 365–390. 2022.

[19] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[20] Songtao Fu, Ke Xu, Qi Li, Xiaoliang Wang, Su Yao, Yangfei Guo, and Xinle Du. Mask: Practical source and path verification based on multi-as-key. In *IWQoS*, pages 1–10, 2021.

[21] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[22] Mustafa Maad Hamdi, Sami AbdulJabbar Rashid, Mahamod Ismail, Mohammed A. Altahrawi, Mohd Fais Mansor, and Mohammed K. Abu-Foul. Performance evaluation of active queue management algorithms in large network. In *ISTT*, pages 1–6, 2018.

[23] Dan Harkins and Dave Carrel. The internet key exchange (ike). Technical report, 1998.

[24] Anxiao He, Kai Bu, Yucong Li, Eikoh Chida, Qianping Gu, and Kui Ren. Atomos: Constant-size path validation proof. *IEEE Transactions on Information Forensics and Security*, 15:3832–3847, 2020.

[25] Anxiao He, Xiang Li, Jiandong Fu, Haoyu Hu, Kai Bu, Chenlu Miao, and Kui Ren. Hummingbird: Dynamic path validation with hidden equal-probability sampling. *IEEE Transactions on Information Forensics and Security*, 18:1268–1282, 2023.

[26] Charles Hornig. A standard for the transmission of ip datagrams over ethernet networks. Technical report, 1984.

[27] IETF. Path aware networking rg (panrg). https://datatracker.ietf.org/rg/panrg/about/, 2019.

[28] Intel. Dpdk: Data plane development kit. http://dpdk.org/.

[29] Intel. Intel xeon processor e5-2630 v3. https://www.intel.com/content/www/us/en/products/sku/83356/intel-xeon-processor-e52630-v3-20m-cache-2-40-ghz/specifications.html.

[30] IRTF. Path aware networking research group panrg. https://irtf.org/panrg, 2019.

[31] Jonathan Katz and Andrew Lindell. Aggregate message authentication codes. *Topics in Cryptology–CT-RSA 2008*, pages 155–169, 2008.

[32] Stephen Kent, Charles Lynn, and Karen Seo. Secure border gateway protocol (s-bgp). *IEEE Journal on Selected areas in Communications*, 18(4):582–592, 2000.

[33] Tiffany Hyun-Jin Kim, Cristina Basescu, Limin Jia, Soo Bum Lee, Yih-Chun Hu, and Adrian Perrig. Lightweight source authentication and path validation. In *SIGCOMM*, volume 44, pages 271–282, 2014.

[34] Jonghoon Kwon, Juan A García-Pardo, Markus Legner, François Wirz, Matthias Frei, David Hausheer, and Adrian Perrig. Scionlab: A next-generation internet testbed. In *ICNP*, pages 1–12, 2020.

[35] Chen-Wei Lee, Chu-Sing Yang, and Yih-Ching Su. Adaptive uep and packet size assignment for scalable video transmission over burst-error channels. *EURASIP Journal on Advances in Signal Processing*, 2006:1–9, 2006.

[36] Markus Legner, Tobias Klenze, Marc Wyss, Christoph Sprenger, and Adrian Perrig. Epic: Every packet is checked in the data plane of a path-aware internet. In *USENIX Security Symposium*, pages 541–558, 2020.

[37] Matthew Luckie and Ben Stasiewicz. Measuring path mtu discovery behaviour. In *IMC*, pages 102–108, 2010.

[38] Robert Lychev, Michael Schapira, and Sharon Goldberg. Rethinking security for internet routing. *Communications of the ACM*, 59(10):48–57, 2016.

[39] Jad Naous, Michael Walfish, Antonio Nicolosi, David Mazières, Michael Miller, and Arun Seehra. Verifying and enforcing network paths with icing. In *CoNEXT*, pages 1–12, 2011.

[40] Vern Paxson. End-to-end routing behavior in the internet. In *SIGCOMM*, pages 25–38, 1996.

[41] Jeffrey Rott. Intel advanced encryption standard instructions (aes-ni). *Technical Report, Technical Report, Intel*, 2010.

[42] Arnab Roy, Anupam Datta, Ante Derek, John C Mitchell, and Jean-Pierre Seifert. Secrecy analysis in protocol composition logic. In *ASIAN*, pages 197–213, 2007.

[43] Binanda Sengupta, Yingjiu Li, Kai Bu, and Robert H Deng. Privacy-preserving network path validation. *ACM Transactions on Internet Technology*, 20(1):1–27, 2020.

[44] Mohammad Sharifkhani and Manoj Sachdev. Sram cell stability: A dynamic perspective. *IEEE Journal of Solid-State Circuits*, 44(2):609–619, 2009.

[45] Yixin Sun, Maria Apostolaki, Henry Birge-Lee, Laurent Vanbever, Jennifer Rexford, Mung Chiang, and Prateek Mittal. Securing internet applications from routing attacks. *Communications of the ACM*, 64(6):86–96, 2021.

[46] Ranysha Ware, Matthew K Mukerjee, Srinivasan Seshan, and Justine Sherry. Modeling bbr's interactions with loss-based congestion control. In *IMC*, pages 137–143, 2019.

[47] Dan Wendlandt, Ioannis Avramopoulos, David G Andersen, and Jennifer Rexford. Don't secure routing protocols, secure data delivery. In *HotNets*, pages 7–12, 2006.

[48] Bo Wu, Ke Xu, Qi Li, Zhuotao Liu, Yih-Chun Hu, Martin J Reed, Meng Shenk, and Fan Yang. Enabling efficient source and path verification via probabilistic packet marking. In *IWQoS*, pages 1–10, 2018.

[49] Fan Yang, Ke Xu, Qi Li, Rongxing Lu, Bo Wu, Tong Zhang, Yi Zhao, and Meng Shen. I know if the journey changes: Flexible source and path validation. In *IWQoS*, pages 1–6, 2020.

[50] Fuyuan Zhang, Limin Jia, Cristina Basescu, Tiffany Hyun-Jin Kim, Yih-Chun Hu, and Adrian Perrig. Mechanized network origin and path authenticity proofs. In *CCS*, pages 346–357, 2014.