SwiftParade: Anti-burst Multipath Validation

Anxiao He, Kai Bu, *Member, IEEE,* Jiongrui Huang, Yifei Pang, Qianping Gu, and Kui Ren, *Fellow, IEEE*

Abstract—Path validation promises a necessary security add-on for future Internet architectures. It authenticates not only source identities but also the exact path where a packet forwards through. This offers users more flexibility and reliability in network services. Most existing solutions focus on single-path validation that pre-correlates a packet to a specific forwarding path. However, parallel transmissions in multipath routing tend to induce bursty traffic that is hardly validated in time by existing solutions. In this paper, we present SwiftParade as the first attempt toward anti-burst multipath validation. It proposes an aggregate validation technique that can simultaneously validate a group of packets likely from multiple different paths. This helps to amortize the validation overhead across packets of the entire group instead of imposing the validation overhead equally on every packet. To implement aggregate validaiton, SwiftParade further explores a noncommutative homomorphic asymmetric encryption scheme. We prove effectiveness and security of SwiftParade through theoretical analysis. We also conduct extensive experiments to evaluate SwiftParade performance. The results show that SwiftParade offers high efficiency and applicability to multipath validation. It outperforms the state-of-the-art multipath validation solution by about an order of magnitude faster validation.

Index Terms—Multipath validation, dynamic routing, anti-burst packet processing.

1 Introduction

Path validation is envisioned to secure packet forwarding. The forwarding process of packets is found to be easily disrupted by various known attacks such as BGP hijacking [1], path re-routing [2], and packet alteration [3], [4]. In the current Internet, however, security of packet forwarding can hardly be validated due to two facts [5]. One is that end-hosts have little control of packet forwarding paths. The other is that the forwarding process leaves little information for end-hosts to verify the exact forwarding trace. To address these limitations, path validation advocates that routers add proofs into packets they forward. Such proofs enable end-hosts to verify whether a packet has traversed the intended forwarding path. A plethora of path validation solutions have been proposed in the recent decade [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16]. As a security add-on, some path validations solutions are integrated into future Internet architecture proposals such as NIRA [17], NEBULA [18], and SCION [19], [20]. A recent path validation representative—EPIC [12]—is even deployed in the first path-aware Internet testbed called SCIONLab [21].

Albeit single-path validation has been extensively investigated, few solutions target multipath routing. In a network supporting multipath routing, two end-hosts can simultaneously use multiple forwarding paths for communication. This not only improves throughput but also improves reliability if redundant packets are sent along different paths [22]. Initial multipath routing protocols require that the set of designated paths be fixed [23]. However, once network status affects the stability of any such fixed path, packet losses might occur. Protocols like equal-cost multipath (ECMP) routing [24] and FatPaths [25] are thus proposed to embrace network dynamics. Multipath routing also supports both packet-grained and flow-grained traffic allocation to satisfy different demands. For example, TeXCP [26], COPE [27], FLARE [28], and LetFlow [29] follow packet granularity and T-RAT [30] characterizes flow granularity.

A key challenge for multipath validation is thus how to efficiently embody a large set of proofs for the allowed paths in packets. Traditional single-path validation solutions only deal with a single source pre-indicated path [6], [7], [8], [9], [10], [11], [12], [13], [14]. However, in multipath routing, packets could be forwarded through any one of designated paths. It means that adopting traditional path validation solutions would generate a large number of path proofs. This can easily lead to heavy computation and communication overheads. Atlas [15] improves efficiency by a hierarchical proof scheme. It helps to limit the proof for each path segment be computed only once, regardless of how many paths a segment may pertain to.

However, we identify another fundamental challenge for multipath validation—flow-integrity violation by burst arrivals. In multipath routing, it is a norm that a large volume of traffic from multiple paths simultaneously arrives at an assembly router. Such burst arrivals tend to exhaust the assembly router's queue capacity and lead to packet losses. We in Section 2.1 qualitatively analyze the impact of multipath routing on packet losses using the state-of-the-art multipath validation solution—Atlas [15]. If no packet loss is allowed, even a 4-path multipath routing policy can limit Atlas bandwidth to as low as 5.83 Gbps.

In this paper, we take on the challenge and propose

- *A. He, K. Bu*, and J. Huang are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China, and are also with ZJU-Hangzhou Global Scientific and Technological Innovation Center, Hangzhou 311215, China.*
  E-mail: {zjuhax, kaibu, jiongrui_huang}@zju.edu.cn
- *Y. Pang is with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China.*
  E-mail: yf.pang@zju.edu.cn
- *Q. Gu is with the School of Computing Science, Simon Fraser University, Burnaby, British Columbia V5A 1S6, Canada.*
  E-mail: qgu@sfu.ca
- *K. Ren is with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China, and is also with the Zhejiang Provincial Key Laboratory of Blockchain and Cyberspace Governance, Hangzhou 310027, China.*
  E-mail: kuiren@zju.edu.cn

**Corresponding Author: Kai Bu.*

SwiftParade as the first anti-burst multipath validation solutions. The key idea is to use a non-commutative homomorphic asymmetric cryptographic scheme for aggregate validation of a set of packets over multipath. Specifically, it supports that a router simultaneously verifies packets received from multiple predecessor routers. The number and order of the packets involved in the verification do not affect the effectiveness of the verification. Furthermore, we propose a two-factor encryption scheme to defend against the second residual module attack [31], which can evade validation by exploiting a vulnerability of ElGamal encryption [32] used in our algorithm.

We implement SwiftParade using DPDK [33] and evaluate its performance. Evaluation results show that SwiftParade is better suited to complex multipath routing in comparison with existing solutions. Specifically, SwiftParade guarantees a constant proof size of 406 bytes. In contrast, Atlas [15] enforces a longer proof when the multipath topology is more complex than one with 4 paths and 6 hops per path. The proof construction and verification times of SwiftParade can be faster than that of Atlas by 148.19× and 8.28×, respectively.

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

- We identify a vulnerability of packet loss due to burst arrivals in multipath validation.
- We propose SwiftParade as the first solution toward anti-burst multipath validation. It gains a leap of efficiency by aggregation validation. Packets from different paths can be simultaneously validated.
- We propose a two-factor encryption scheme to defend against the second residual module attack.
- We implement SwiftParade using DPDK and evaluate its performance through extensive experiments. SwiftParade offers a high communication efficiency via a constant-size proof regardless of path length and path number. The aggregate validation technique enables SwiftParade outperform the state-of-the-art multipath validation solution in terms of an order of magnitude faster validation.

The rest of the paper is organized as follows. Section 2 identifies the vulnerability of bursty traffic to multipath validation and motivates an aggregate validation technique as mitigation. Section 3 presents SwiftParade as the first solution for anti-burst multipath validation. Section 4 details the design strategies. Section 5 and Section 6 prove SwiftParade security and evaluate its performance in comparison with the state-of-the-art solution, respectively. Section 7 reviews and distinguishes related work from SwiftParade. Finally, Section 8 concludes the paper.

2 Motivation

In this section, we identify an intrinsic challenge for multipath validation. Parallel transmissions in multipath routing tend to induce burst arrivals on converging routers. Without being validated fairly fast, such bursty traffic may overflow input queues and cause packet losses. This motivates us to explore an aggregate validation technique that can simultaneously validate packets from different incoming paths. Aggregate validation offers a leap of efficiency by no longer trapping in a single packet-path binding per validation operation as in existing solutions.

2.1 Packet Loss upon Burst Arrivals

We perform a qualitative analysis of the susceptibility of packet loss due to burst arrivals in multipath validation. In particular, we focus on potential packet losses on converging routers due to burst transmissions. A packet loss occurs when the input queue on a router is full and can hold no more incoming packets. Given a queue holding up to $\mathcal{Q}$ packets, whether the number $\mathcal{Q}'$ of incoming packets exceeds queue capacity depends on how fast the router processes a packet. Let $\mathcal{T}$ denote the processing time per packet. The faster $\mathcal{T}$ can be, the fewer packets can be stuck in the queue before the router processes them. During the time span of $\mathcal{T}$, the number of newly arrived packets along a single path is $\frac{\mathcal{B}}{\mathcal{S}} \times \mathcal{T}$, where $\mathcal{B}$ and $\mathcal{S}$ represent bandwidth of a path and size of a packet, respectively. Now let us take into account of the multipath effect. Given a number $\mathcal{N}$ of paths for multipath routing toward a converging router, we can derive the maximal number $\mathcal{P}$ of incoming packets while the router processes a packet in $\mathcal{T}$ as the following:

$$\mathcal{P} = \frac{\mathcal{B}}{\mathcal{S}} \times \mathcal{T} \times \mathcal{N}.$$

Definition 1. ***Packet loss susceptibility*** Δ**:** *Given queue size $\mathcal{Q}$, processing time $\mathcal{T}$, path bandwidth $\mathcal{B}$, packet size $\mathcal{S}$, and path number $\mathcal{N}$, a converging router is susceptible to packet loss if the indicator of packet loss susceptibility Δ is set as 1. We define Δ as follows.*

$$\Delta = \begin{cases} 0, & \text{if } \mathcal{Q} - \mathcal{P} = \mathcal{Q} - \frac{\mathcal{B}}{\mathcal{S}} \times \mathcal{T} \times \mathcal{N} \geq 0, \\ 1, & \text{if } \mathcal{Q} - \mathcal{P} = \mathcal{Q} - \frac{\mathcal{B}}{\mathcal{S}} \times \mathcal{T} \times \mathcal{N} < 0. \end{cases} \quad (1)$$

By Equation 1, packet loss susceptibility increases with processing time, path bandwidth, and path number while decreasing with packet size. Figure 1 shows the number of newly arrived 1000-byte packets with varying processing time, path bandwidth, and path number. Consider a common queue size of 140 [34]. Queue size $\mathcal{Q}$ allocated to $\mathcal{N}$ paths ranges from 140 to 140$\mathcal{N}$ depending on granularity of queue sharing among input ports. 140 corresponds to an extreme case when all the $\mathcal{N}$ input ports share the same queue. 140$\mathcal{N}$ lies in the other extreme where each input port features with an individual queue. When the number of newly arrived packets in Figure 1 exceeds $\mathcal{Q}$, packet losses may occur. In other words, queue size $\mathcal{Q}$ puts an implicit limit on bandwidth. Consider Atlas [15]—state-of-the-art multipath validation—for example. It takes about 6 μs to process a packet. Queue size of 140∼140$\mathcal{N}$ limits Atlas bandwidth as 23.33 Gbps (Figure 1(a) when $\mathcal{N} = 1$), 11.67∼23.33 Gbps (Figure 1(b) when $\mathcal{N} = 2$), 5.83∼23.33 Gbps (Figure 1(c) when $\mathcal{N} = 4$), and 3.89∼23.33 Gbps (Figure 1(d) when $\mathcal{N} = 6$). This means that Atlas can only guarantee non-occurrence of packet loss with bandwidth below 23.33 Gbps when $\mathcal{N} = 1$, below 11.67 Gbps when $\mathcal{N} = 2$, below 5.83 Gbps when $\mathcal{N} = 4$, and below 3.89 Gbps when $\mathcal{N} = 6$. Once bandwidth exceeds these thresholds, packet losses become possible; packet losses are bound to happen when bandwidth exceeds 23.33 Gbps. Since the thresholds for

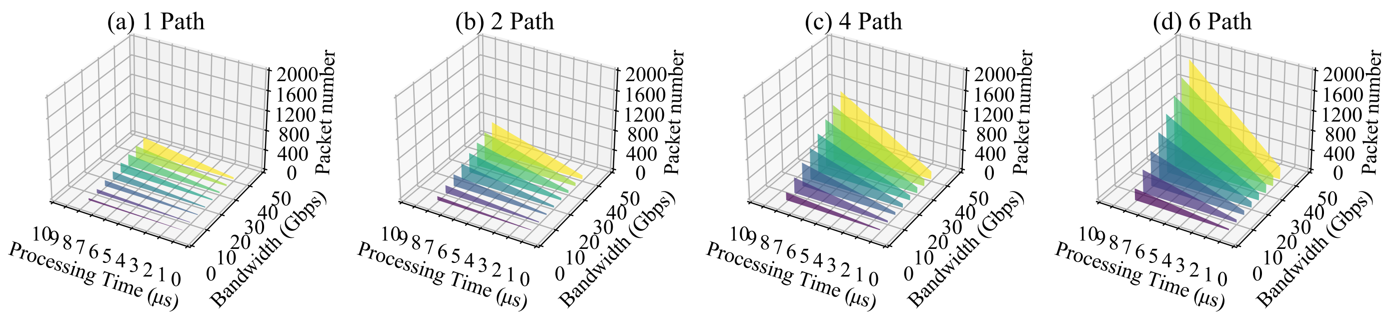

Fig. 1: Number of 1,000-byte packets received by a router with varying processing time, bandwidth, and path number.

packet-loss likability decrease with path number, multipath validation is more susceptible to packet posses due to burst arrivals.

2.2 Aggregate Validation

To make multipath validation robust against burst arrivals, we are motivated to explore aggregate validation. We expect that a feasible aggregate validation technique can simultaneously validate a group of packets instead of an individual one. It should satisfy the following three properties in practical networks using multipath routing.

Property 1: Inclusive validation. Existing path validation solutions enforce exclusive validation in that they process only a single packet per operation. In contrast, aggregate validation advocates an inclusive fashion. Such inclusiveness shows in two aspects. First, a validation operation is no longer limited to only a single packet. It takes effect over a group of packets. Second, the group of packets under validation need not be limited to the same forwarding path. They could have arrived from different incoming paths.

Property 2: Dynamic routing. We need to embrace also flexibility of path choices in multipath routing. A typical line of multipath routing protocols dynamically splits traffic across allowed paths at each hop [26], [27], [28], [29], [30]. This optimizes usage of all available path resources and parallelizes traffic transmission in order to minimize the propagation delay of the entire group of traffic. However, existing single-path validation solutions assume a fixed packet-path binding along the entire forwarding process. They can hardly be adapted to efficient multipath validation. The state-of-the-art multipath validation—Atlas [15]—still uses single-path validation primitives (i.e., OPT [7]). It essentially treats multipath validation as a combination of single-path validation and optimizes efficiency by, for example, avoiding repeated validation of overlapping segments over different paths.

Property 3: Constant-size proof. From the implementation point of view, validation proofs should be crafted to ease aggregate validation. Most existing path validation solutions use proofs with size linearly increasing with path length. Following the wisdom in various cryptography algorithms, it might be quite challenging to simultaneously operate on a group of proofs with different lengths. We thus explore cryptography primitives to offer constant-size proofs regardless of path length.

We design an aggregate validation technique with all the preceding properties satisfied and implement it through SwiftParade. Table 1 compares SwiftParade with both single-path and multipath validation solutions. SwiftParade stands for the first solution that supports aggregate validation. Our aggregate validation can efficiently and simultaneously validate a group of packets likely from different incoming paths. It enables SwiftParade to suit more for anti-burst multipath validation.

TABLE 1: Comparison of properties that different solutions satisfy ✓, partially satisfy ✓*, and unsatisfy ✗.

Solution	Scenario	Property		
	Multipath Validation	Inclusive Validation	Dynamic Routing	Constant-Size Proof
ICING [6]	✗	✗	✗	✗
OPT [7]	✗	✗	✗	✗
OSV [8], [9]	✗	✗	✗	✗
PPV [10]	✗	✗	✗	✓
Atomos [11]	✗	✗	✗	✓
EPIC [12]	✗	✗	✗	✗
PSVM [13]	✗	✗	✓*	✗
MASK [14]	✗	✗	✗	✓
Atlas [15]	✓	✗	✓	✗
VALNET [16]	✓	✗	✓	✗
SwiftParade	✓	✓	✓	✓

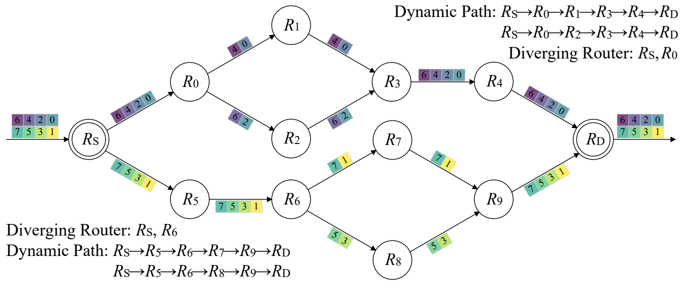
Dynamic Path: $R_S \to R_0 \to R_1 \to R_3 \to R_4 \to R_D$
$R_S \to R_0 \to R_2 \to R_3 \to R_4 \to R_D$
Diverging Router: R_S, R_0
R_S
R_0
R_1
R_2
R_3
R_4
R_5
R_6
R_7
R_8
R_9
R_D
6 4 2 0
7 5 3 1
Diverging Router: R_S, R_6
Dynamic Path: $R_S \to R_5 \to R_6 \to R_7 \to R_9 \to R_D$
$R_S \to R_5 \to R_6 \to R_8 \to R_9 \to R_D$

Fig. 2: Example topology of multipath routing.

3 Overview

In this section, we first define the system and adversary models for SwiftParade. We then highlight the key design principles for anti-burst multipath validation. We will detail the design strategies in Section 4.

3.1 System Model

Figure 2 illustrates the system model of our anti-burst multipath validation scheme. Without loss of generality, we consider an excerpt topology out of a larger network that applies multipath routing. The topology of interest offers four paths from the source R_S to the destination R_D.

Packet transmission along the four paths is parallel and dynamic toward optimizing throughput [26], [27], [28], [29], [30]. Such two features distinguish multipath routing from single-path routing and necessitate the anti-burst requirement. First, parallel transmission requires that diverging routers forward received packets along all available paths toward the destination. Diverging routers in Figure 2 are R_S, R_0, and R_6. For example, R_S forwards the eight received packets along two available paths through two 4-packet transmissions. Note that packet-to-path allocation need not be even; it usually depends on link status of available paths [26], [27], [28], [29], [30]. Second, dynamic transmission requires that the forwarding decision of a packet be randomly determined at each router. In other words, no fixed packet-path binding is enforced. This still helps to adapt packet transmission to link status and maximizes multipath effect.

For ease of presentation, we assume that all on-path routers have incorporated path validation functionalities. Practical deployment usually enforces path validation on vantage-point routers (e.g., SCIONLab [21]).

3.2 Adversary Model

As with the literature [12], we consider an active attacker with capabilities constrained by the Dolev-Yao model [35]. Such an attacker can not only passively monitor network traffic but also actively drop, deviate, alter, and inject packets. It gains these capabilities through, for example, hijacking or compromising routers. The ultimate attack goal against path validation is to succeed validation even if a packet has not followed a valid path. Once the attacker succeeds, it may undesirably escalate service quality or evade security enforcement [5], [6], [7], [12]. A successful attack requires the attacker to forge a valid proof. The adopted cryptographic primitives are public to both routers and potential attackers. Secret data such as private keys are kept proprietary to only their owning routers.

4 Design

In this section, we detail and prove SwiftParade design. The workflow lies in three key algorithms—`Initialization`, `Construction`, and `Verification`. `Initialization` sets up path validation via, for example, key establishment and exchange among routers. Then the source follows `Construction` to generate packet proofs and embed them into packet headers before sending packets out. Upon receiving packets, a router first invokes `Verification` for constructing and validating the aggregate proof of a group of packets. It then calls `Construction` to integrate its own credentials into each validated packet.

4.1 SwiftParade Principle

We develop a noncommutative homomorphic asymmetric encryption scheme to fulfill aggregate validation. It is inspired by Atomos that leverages noncommutative homomorphic asymmetric encryption toward a constant-size proof [11]. Our SwiftParade takes another leap of the aggregation effect. It not only aggregates proofs from co-path routers into a constant-size one but also aggregates such constant-size proofs from a group of packets along different paths into one that validates forwarding correctness of all the grouped packets. Notations and abbreviations for ease of detailing SwiftParade are summarized in Table 2.

TABLE 2: Definition of notations and abbreviations.

Notation	Definition
S	source node, that is, R_S
D	destination node, that is, R_D
Φ	path consisting of routers $(R_S, ..., R_i, ..., R_D)$
P	the packet involving in path validation
R_i	on-path router
A_i	a set of predecessor routers of R_i
r_i	a set of aggregated packets on R_i
m	the number of packets aggregated on R_i
p	the prime used for modulo
g	the generator of p
x_i	private key of R_i
y_i	public key of R_i
$SessionID$	identifier of a session
$Timestamp$	creation time of the packet
$DataHash$	hash of the packet payload
σ_i	proof field for R_i to deliver multisignature
ρ_i	proof field for R_i to deliver forwarding order
c_i	random number selected by R_i
u_i	auxiliary parameter generated by R_i for verification
$H(\cdot)$	cryptographic hash function
$\pi(\cdot)$	recursive function to deliver validation parameters
a_i	identifier of router R_i

Noncommutative homomorphic asymmetric encryption. We first recap necessary definitions for noncommutative homomorphic asymmetric encryption proposed in [11]. Of our major interest are two magmas (additive and multiplicative) and a noncommutative homomorphic mapping function constructed using the two magmas. Let $p > 2$ denote a large prime number. Then we define $\mathbb{Z}_p = \{0, 1, ..., p-1\}$ and $\mathbb{Z}_p^* = \{1, 2, ..., p-1\}$.

Definition 2. ***[11] Additive Magma*** $(\mathbb{Z}_{p-1} \times \mathbb{Z}_{p-1}, \star)$*: We define a binary operation* $\star$ *as:*

$$(a_1, b_1) \star (a_2, b_2) = (a_1 + a_2, a_1 + b_1 + b_2) \bmod (p-1).$$

Definition 3. ***[11] Multiplicative Magma*** $(\mathbb{Z}_p^* \times \mathbb{Z}_p^*, \odot)$*: We define a binary operation* $\odot$ *as:*

$$(a_1, b_1) \odot (a_2, b_2) = (a_1 a_2, a_1 b_1 b_2) \bmod p.$$

Definition 4. ***[11] Noncommutative Homomorphic Mapping Function*** $F : (\mathbb{Z}_{p-1} \times \mathbb{Z}_{p-1}, \star) \rightarrow (\mathbb{Z}_p^* \times \mathbb{Z}_p^*, \odot)$*: We define a function* F *as follows:*

$$F(a, b) = (f(a), f(b)) = (g^a, g^b),$$

where the function $f : \mathbb{Z}_{p-1} \rightarrow \mathbb{Z}_p^*$ *is defined as* $f(x) = g^x \bmod p$ *and* g *is a generator of* $\mathbb{Z}_p^*$.

Using the preceding definitions, one can construct the basic noncommutative homomorphic asymmetric encryption scheme as follows [11].

$$F((a_1, b_1) \star (a_2, b_2)) = F(a_1, b_1) \odot F(a_2, b_2).$$

Aggregate validation. We now present our construction of noncommutative homomorphic asymmetric encryption that supports aggregate validation. The construction embodies the following three designs. Note that we sketch only key design principles here and provide more implementation details in Section 4.

- *First, how to construct packet proofs for inclusive validation?*
  We integrate path flexibility into a constant-size packet proof. More specifically, when router R_i computes its proof $(\tilde{\sigma}_i, \tilde{\rho}_i)$ for a packet, it no longer simply considers the only previous hop the packet has traversed. Instead, R_i takes into account all the allowed previous hops (denoted as set A_i) for the packet. The construction of $(\tilde{\sigma}_i, \tilde{\rho}_i)$ is as follows:
  $$(\tilde{\sigma}_i, \tilde{\rho}_i) = (\tilde{\sigma}_{A_i}, \tilde{\rho}_{A_i}) \star (\sigma_i, \rho_i), \quad (2)$$
  where $\tilde{\sigma}_{A_i} = \sum_{a\in A_i} \tilde{\sigma}_a$ and $\tilde{\rho}_{A_i} = \sum_{a\in A_i} \tilde{\rho}_a$.
- *Second, how to aggregate proofs over a group of packets?*
  Let $(\sigma^j_{A_i}, \rho^j_{A_i})$ denote the proof of the jth packet in a set r_i of m packets on router R_i. We aggregate proofs over the set of packets as follows.
  $$(\tilde{\sigma}_{r_i}, \tilde{\rho}_{r_i}) = (\tilde{\sigma}_{r_{A_i}}, \tilde{\rho}_{r_{A_i}}) \star (\sigma_{r_i}, \rho_{r_i}), \quad (3)$$
  where
  $$\tilde{\sigma}_{r_{A_i}} = \sum_{j=1}^{m} \tilde{\sigma}^j_{A_i},$$
  $$\tilde{\rho}_{r_{A_i}} = \sum_{j=1}^{m} \tilde{\rho}^j_{A_i},$$
  $$\sigma_{r_i} = \sum_{j=1}^{m} \sigma^j_{A_i}, \rho_{r_i} = \sum_{j=1}^{m} \rho^j_{A_i}.$$
- *Third, how to verify the aggregate proof to simultaneously validate forwarding correctness of the entire set of packets?*
  Following the preceding construction principles and subsequent design specifics in Section 4, we validate an aggregated proof $(\tilde{\sigma}_{r_i}, \tilde{\rho}_{r_i})$ if it satisfies the following equation.
  $$F(\tilde{\sigma}_{r_i}, \tilde{\rho}_{r_i}) = F(\tilde{\sigma}_{r_{A_i}}, \tilde{\rho}_{r_{A_i}}) \odot F(\sigma_{r_i}, \rho_{r_i}). \quad (4)$$
  We will prove the correctness of our validation scheme in Section 4.4.

4.2 Initialization

`Initialization` aims to establish and exchange keys among routers (Algorithm 1). It first initializes a modulo and a generator for function f used by the noncommutative homomorphic mapping function in Definition 4 (lines 3-5). Note that we use a composite number $N = p' \times p''$ rather than a single prime number to defend against a second residual module attack (Section 5.3). Similarly, we select two large prime numbers—x'_i and x''_i respectively from $\mathbb{Z}_{p'-1}$ and $\mathbb{Z}_{p''-1}$—as private keys of router R_i (line 8). Then we compute the public key y_i of router R_i as $y_i = f(x'_i + x''_i) = g^{x'_i+x''_i} \bmod N$ (line 9) by Definition 4. Following the design strategies of asymmetric key encryption, routers keep their secret keys privately and exchange identifiers and public keys with each other (line 11).

Algorithm 1: SwiftParade `Initialization`

```
1  Function Initialization:
2      //step 1: generator generation
3      $p', p'' \leftarrow$ large random prime numbers;
4      $g \leftarrow$ a generator of $p'$;
5      $N = p'p''$;
6      //step 2: key generation
7      for each router $R_i$ do
8          $x'_i, x''_i \leftarrow$ random prime numbers;
9          $y_i \leftarrow g^{x'_i+x''_i} \bmod N$;
10     //step 3: key exchange
11     Router $R_i$ sends $(R_i, y_i)$ to other routers;
```

Algorithm 2: SwiftParade `Construction`

```
1  Function Construction:
2      if router $R_i$ is the source then
3          SessionID $\leftarrow$ identifier of the current session;
4          Timestamp $\leftarrow$ creation time of the packet with payload $P$;
5          DataHash $\leftarrow H(P)$;
6          //construction of SwiftParade Proof
7          $h = \text{DataHash}||\text{SessionID}||\text{Timestamp}$;
8          $G = H(\text{SessionID})$;
9          $c'_1, c''_1 \leftarrow$ random numbers from $\mathbb{Z}_{p'-1}, \mathbb{Z}_{p''-1}$;
10         $\sigma'_1 \leftarrow x'_1 + c'_1(\prod_{j=1}^{m} H(h)^j)G \bmod (p'-1)$;
11         $\sigma''_1 \leftarrow x''_1 + c''_1(\prod_{j=1}^{m} H(h)^j)G \bmod (p''-1)$;
12         $\rho_1 = \sigma_1 \leftarrow \sigma'_1 + \sigma''_1$;
13         $(\tilde{\sigma}_1, \tilde{\rho}_1) \leftarrow (\sigma_1, \rho_1)$;
14         $u_1 \leftarrow g^{(c'_1+c''_1)(\prod_{j=1}^{m} H(h)^j)} \bmod N$;
15         $\pi(u_1) = \tilde{u}_1 \leftarrow u_1$;
16         $\text{Proof}_1 \leftarrow (\tilde{\sigma}_1, \tilde{\rho}_1)||\tilde{u}_1||\pi(u_1)$;
17     else
18         $h = \text{DataHash}||\text{SessionID}||\text{Timestamp}$;
19         $G = H(\text{SessionID})$;
20         $c'_i, c''_i \leftarrow$ random numbers from $\mathbb{Z}_{p'-1}, \mathbb{Z}_{p''-1}$;
21         $\sigma'_i \leftarrow x'_i + c'_i(\prod_{j=1}^{m} H(h)^j)G \bmod (p'-1)$;
22         $\sigma''_i \leftarrow x''_i + c''_i(\prod_{j=1}^{m} H(h)^j)G \bmod (p''-1)$;
23         $\rho_i = \sigma_i \leftarrow \sigma'_i + \sigma''_i$;
24         $\tilde{\sigma}_{A_i} = \sum_{a\in A_i} \tilde{\sigma}_a$;
25         $\tilde{\rho}_{A_i} = \sum_{a\in A_i} \tilde{\rho}_a$;
26         $(\tilde{\sigma}_i, \tilde{\rho}_i) \leftarrow (\tilde{\sigma}_{A_i}, \tilde{\rho}_{A_i}) \star (\sigma_i, \rho_i)$;
27         $u_i = g^{(c'_i+c''_i)(\prod_{j=1}^{m} H(h)^j)} \bmod N$;
28         $\tilde{u}_i \leftarrow (\prod_{a\in A_i} \tilde{u}_a)u_i \bmod N$;
29         $\pi(u_i) \leftarrow (\prod_{a\in A_i} \pi(u_a))\tilde{u}_i \bmod N$;
30         $\text{Proof}_i \leftarrow (\tilde{\sigma}_i, \tilde{\rho}_i)||\tilde{u}_i||\pi(u_i)$;
```

4.3 Construction

As shown in Algorithm 2, `Construction` follows different logics on the source (lines 2-16) and routers (lines 18-30) for computing path proofs. This is mainly because that the source has no previous hops to consider for inclusive validation. Note also that the destination no longer needs to construct proofs; it only needs to verify proofs as `Verification` instructs in Algorithm 3.

Proof construction at the source. Along with the path proof, the source also needs to embed SessionID, Timestamp, and DataHash into the packet header (lines 3-5). These fields ease for correlating co-session packets and verifying packet freshness and integrity [7], [11]. Of particular emphasis is SessionID that routers use to locally correlate packets in a session with their designated forwarding paths [7],

[11]. It also helps to quickly filter cetain mis-forwarded packets whose SessionID contradicts with the actual path they traverse.

The path proof built by SwiftParade requires a two-tuple—(σ_i, ρ_i) (lines 6-16). It follows aggregate computation in that we construct the path proof using metadata from an entire group of packets and then update the constructed proof into each packet.

- The first item σ_i is used to prove the router identity. Therefore, the computation of σ_i uses router R_i's secret key so that it can be verified by other routers using R_i's public key. Specifically, the source R_1 computes σ_1 as follows (lines 10-12).

$$\sigma_1' = x_1' + c_1'(\prod_{j=1}^{m} H(h)^j)G \bmod (p'-1),$$
$$\sigma_1'' = x_1'' + c_1''(\prod_{j=1}^{m} H(h)^j)G \bmod (p''-1),$$
$$\sigma_1 = \sigma_1' + \sigma_1'',$$

  where $h = \text{DataHash}||\text{SessionID}||\text{Timestamp}$ guarantees packet integrity and distinguishes packets with different proofs, $(\prod_{j=1}^{m} H(h)^j)$ denotes the aggregation of different packets' identity and $G = H(\text{SessionID})$ acts as a fixed parameter for aggregating different co-session packets.
- The second item ρ_i is used to track the forwarding order along routers. It follows the same computation process as σ_i (line 12). However, by integrating ρ_i into the additive magma defined in Definition 2, the noncommutativity property of our encryption scheme guarantees that mis-forwarded packets fail validation and get filtered.

We now augment the proof with two more fields—$\tilde{u}_i$ and $\pi(u_i)$—for ease of verification on subsequent routers. Without these two fields, subsequent routers cannot verify (σ_1, ρ_1) because they relate to unknown random numbers c_1' and c_1''. We address this challenge by rendering c_1' and c_1'' as a one-time secret key in the context of asymmetric cryptography. Then we compute the corresponding public key u_1 as follows (line 14).

$$u_1 = g^{(c_1'+c_1'')(\prod_{j=1}^{m} H(h)^j)} \bmod N,$$

where $H(h)$ is included to guarantee that u_1 is packet specific as h depends on DataHash. Derived from u_1 are the two auxiliary proof fields $\tilde{u}_1 = u_1$ and $\pi(u_1) = u_1$ (line 15). Both will be discussed shortly about how they help the next-hop router to verify R_1's proof in Definition 5.

Definition 5. ***Proof generated by the Source*** R_1**:** *The proof constructed by the source R_1 is defined as follows:*

$$\text{Proof}_1 = (\tilde{\sigma}_1, \tilde{\rho}_1)||\tilde{u}_1||\pi(u_1),$$

where we have

$$(\tilde{\sigma}_1, \tilde{\rho}_1) = (\sigma_1, \rho_1).$$

Proof construction on intermediate routers. Upon an intermediate router R_i receives a set of packets, it first verifies their aggregate proof (Section 4.4). Then R_i updates validated proofs with its own credentials. The update maintains both the forwarding order and the constant proof size (lines 18-30). First, router R_i constructs its signature pair (σ_i, ρ_i) as follows (lines 21-23).

$$\sigma_i' = x_i' + c_i'(\prod_{j=1}^{m} H(h)^j)G \bmod (p'-1),$$
$$\sigma_i'' = x_i'' + c_i''(\prod_{j=1}^{m} H(h)^j)G \bmod (p''-1),$$
$$\rho_i = \sigma_i = \sigma_i' + \sigma_i''. \quad (5)$$

Then it computes the following field $(\tilde{\sigma}_i, \tilde{\rho}_i)$ by Equation 2 (lines 24-26).

$$\tilde{\sigma}_{A_i} = \sum_{a \in A_i} \tilde{\sigma}_a,$$
$$\tilde{\rho}_{A_i} = \sum_{a \in A_i} \tilde{\rho}_a,$$
$$(\tilde{\sigma}_i, \tilde{\rho}_i) = (\tilde{\sigma}_{A_i}, \tilde{\rho}_{A_i}) \star (\sigma_i, \rho_i),$$

where $\tilde{\sigma}_a$ and $\tilde{\rho}_a$ are inherited from proofs of R_i's predecessor routers.

R_i now continues to compute the two auxiliary fields—$\tilde{u}_i$ and $\pi(u_i)$ as follows (lines 27-29).

$$u_i = g^{(c_i'+c_i'')(\prod_{j=1}^{m} H(h)^j)} \bmod N.$$
$$\tilde{u}_i = (\prod_{a \in A_i} \tilde{u}_a)u_i \bmod N.$$
$$\pi(u_i) = (\prod_{a \in A_i} \pi(u_a))\tilde{u}_i \bmod N.$$

Definition 6. ***Proof generated by intermediate router*** R_i**:** *The proof constructed by an intermediate router R_i is defined as follows:*

$$\text{Proof}_i = (\tilde{\sigma}_i, \tilde{\rho}_i)||\tilde{u}_i||\pi(u_i).$$

Algorithm 3: SwiftParade `Verification`

```
1  Function Verification:
2      //AggregateProof$_{r_i}$ is verified by $R_{i+1}$;
3      $\tilde{y}_{r_i} \leftarrow \prod_{j=1}^{m}((\prod_{a \in A_i} \tilde{y}_a)y_i)^j \bmod N$;
4      $\pi(y_{r_i}) \leftarrow \prod_{j=1}^{m}(\prod_{a \in A_i} \pi(y_a)\tilde{y}_i)^j \bmod N$;
5      $\tilde{u}_{r_i} \leftarrow \prod_{a \in r_i} \tilde{u}_a \bmod N$;
6      $\pi(u_{r_i}) \leftarrow (\prod_{a \in r_i} \pi(u_a)) \bmod N$;
7      $left_1 \leftarrow g^{\tilde{\sigma}_{r_i}} \bmod N$;
8      $right_1 \leftarrow g^{\tilde{\rho}_{r_i}} \bmod N$;
9      $left_2 \leftarrow \tilde{y}_{r_i}(\tilde{u}_{r_i})^G \bmod N$;
10     $right_2 \leftarrow \pi(y_{r_i})(\pi(u_{r_i}))^G \bmod N$;
11     if $left_1 == left_2$ && $right_1 == right_2$ then
12         Accept packets and update proofs;
13     else
14         Drop packets;
```

4.4 Verification

`Verification` in Algorithm 3 presents how router R_{i+1} verifies whether an aggregate proof satisfies Equation 4 (Section 4.1):

$$F(\tilde{\sigma}_{r_i}, \tilde{\rho}_{r_i}) = F(\tilde{\sigma}_{r_{A_i}}, \tilde{\rho}_{r_{A_i}}) \odot F(\sigma_{r_i}, \rho_{r_i}).$$

We next prove that our construction of `Verification` satisfies validity test of Equation 4 in Theorem 1. Then we will demonstrate that such validity test can hardly be circumvented by forged proofs in Section 5.

Lemma 1. *The proof* $\mathrm{Proof}_i = (\tilde{\sigma}_i, \tilde{\rho}_i)||\tilde{u}_i||\pi(u_i)$ *of a packet is valid if the following equations hold:*

$$g^{\tilde{\sigma}_i} = \tilde{y}_i(\tilde{u}_i)^G \bmod N, \tag{6}$$

$$g^{\tilde{\rho}_i} = \pi(y_i)(\pi(u_i))^G \bmod N, \tag{7}$$

where $\tilde{y}_i$ *and* $\pi(y_i)$ *are calculated the same as* $\tilde{u}_i$ *and* $\pi(u_i)$*:*

$$\tilde{y}_i = (\prod_{a\in A_i} \tilde{y}_a)y_i \bmod N, \pi(y_i) = (\prod_{a\in A_i} \pi(y_a))\tilde{y}_i \bmod N.$$

Proof. A valid proof should satisfy the following equation (Section 4.1):

$$F(\tilde{\sigma}_i, \tilde{\rho}_i) = F(\tilde{\sigma}_{A_i}, \tilde{\rho}_{A_i}) \odot F(\sigma_i, \rho_i). \tag{8}$$

The left side of this equation can be derived as:

$$F(\tilde{\sigma}_i, \tilde{\rho}_i) = (g^{\tilde{\sigma}_i}, g^{\tilde{\rho}_i}). \tag{9}$$

The right side of this equation can be derived as:

$$\begin{aligned} &F(\tilde{\sigma}_{A_i}, \tilde{\rho}_{A_i}) \odot F(\sigma_i, \rho_i) \\ &= (g^{\tilde{\sigma}_{A_i}}, g^{\tilde{\rho}_{A_i}}) \odot (g^{\sigma_i}, g^{\rho_i}) \\ &= (g^{\tilde{\sigma}_{A_i}+\sigma_i}, g^{\tilde{\sigma}_{A_i}+\tilde{\rho}_{A_i}+\rho_i}). \end{aligned} \tag{10}$$

To derive Equation 10 from Equation 9, we make further derivations for the two items in the tuple. The first item is derived as follows:

$$\begin{aligned} g^{\tilde{\sigma}_{A_i}+\sigma_i} &= (\prod_{a\in A_i} \tilde{y}_a)(\prod_{a\in A_i} \tilde{u}_a)^G y_i(u_i)^G \\ &= \tilde{y}_i(\tilde{u}_i)^G. \end{aligned}$$

The second item is derived as follows:

$$\begin{aligned} g^{\tilde{\sigma}_{A_i}+\tilde{\rho}_{A_i}+\rho_i} &= (\prod_{a\in A_i} \tilde{y}_a)(\prod_{a\in A_i} \tilde{u}_a)^G \\ &(\prod_{a\in A_i} \pi(y_a))(\prod_{a\in A_i} \pi(u_a))^G y_i(u_i)^G \\ &= (\prod_{a\in A_i} \tilde{y}_a)y_i(\prod_{a\in A_i} \tilde{u}_a)^G(u_i)^G \\ &(\prod_{a\in A_i} \pi(y_a))(\prod_{a\in A_i} \pi(u_a))^G \\ &= \tilde{y}_i(\tilde{u}_i)^G(\prod_{a\in A_i} \pi(y_a))(\prod_{a\in A_i} \pi(u_a))^G \\ &= \pi(y_i)(\pi(u_i))^G. \end{aligned}$$

Thus, the right side of Equation 8 can be finally derived as follows.

$$F(\tilde{\sigma}_{A_i}, \tilde{\rho}_{A_i}) \odot F(\sigma_i, \rho_i) = (\tilde{y}_i(\tilde{u}_i)^G, \pi(y_i)(\pi(u_i))^G)$$

From the preceding equations we can observe that the validity of Equation 8 depends on that of Equation 6 and Equation 7. This proves Lemma 1. □

Theorem 1. *The aggregate proof on router* R_{i+1} *(i.e.,* $\mathrm{Proof}_{r_i} = (\tilde{\sigma}_{r_i}, \tilde{\rho}_{r_i})||\tilde{u}_{r_i}||\pi(u_{r_i})$*) is valid (i.e., satisfying Equation 4) if the following equations hold (line 11 in Algorithm 3):*

$$g^{\tilde{\sigma}_{r_i}} = \tilde{y}_{r_i}(\tilde{u}_{r_i})^G \bmod N, \tag{11}$$

$$g^{\tilde{\rho}_{r_i}} = \pi(y_{r_i})(\pi(u_{r_i}))^G \bmod N, \tag{12}$$

where we have

$$\tilde{y}_{r_i} = \prod_{a\in r_i} \tilde{y}_a \bmod N, \tag{13}$$

$$\tilde{u}_{r_i} = \prod_{a\in r_i} \tilde{u}_a \bmod N, \tag{14}$$

$$\pi(y_{r_i}) = (\prod_{a\in r_i} \pi(y_a)) \bmod N, \tag{15}$$

$$\pi(u_{r_i}) = (\prod_{a\in r_i} \pi(u_a)) \bmod N. \tag{16}$$

Proof. A valid proof should satisfy Equation 4 (Section 4.1):

$$F(\tilde{\sigma}_{r_i}, \tilde{\rho}_{r_i}) = F(\tilde{\sigma}_{r_{A_i}}, \tilde{\rho}_{r_{A_i}}) \odot F(\sigma_{r_i}, \rho_{r_i}).$$

By Definition 4, the left side of Equation 4 can be derived as follows (lines 7 and 8 in Algorithm 3).

$$F(\tilde{\sigma}_{r_i}, \tilde{\rho}_{r_i}) = (g^{\tilde{\sigma}_{r_i}}, g^{\tilde{\rho}_{r_i}}). \tag{17}$$

The right side of Equation 4 can be derived as follows.

$$\begin{aligned} &F(\tilde{\sigma}_{r_{A_i}}, \tilde{\rho}_{r_{A_i}}) \odot F(\sigma_{r_i}, \rho_{r_i}) \\ &= (g^{\tilde{\sigma}_{r_{A_i}}}, g^{\tilde{\rho}_{r_{A_i}}}) \odot (g^{\sigma_{r_i}}, g^{\rho_{r_i}}) \\ &= (g^{\tilde{\sigma}_{r_{A_i}}+\sigma_{r_i}}, g^{\tilde{\sigma}_{r_{A_i}}+\tilde{\rho}_{r_{A_i}}+\rho_{r_i}}). \end{aligned} \tag{18}$$

The second line follows Definition 4 and the third line follows Definition 3.

The validity of Equation 4 is now narrowed down to the equality of Formulae 17 and 18. We further derive both items in the tuple of Formula 18 as follows.

$$\begin{aligned} g^{\tilde{\sigma}_{r_{A_i}}+\sigma_{r_i}} &= g^{(\tilde{\sigma}^1_{A_i}+\tilde{\sigma}^1_i)+\ldots+(\tilde{\sigma}^m_{A_i}+\tilde{\sigma}^m_i)} \\ &= g^{\tilde{\sigma}^1_i+\tilde{\sigma}^2_i+\ldots+\tilde{\sigma}^m_i} \\ &= \tilde{y}^1_i(\tilde{u}^1_i)^G\tilde{y}^2_i(\tilde{u}^2_i)^G\ldots\tilde{y}^m_i(\tilde{u}^m_i)^G \qquad (19) \\ &= \tilde{y}_{r_i}(\tilde{u}_{r_i})^G, \qquad (20) \end{aligned}$$

where the first and second lines follow our construction in Section 4.1, Formula 19 follows Lemma 1, and Formula 20 is a simplified form following Equations 13 and 14 (line 9 in Algorithm 3).

$$\begin{aligned} g^{\tilde{\sigma}_{r_{A_i}}+\tilde{\rho}_{r_{A_i}}+\rho_{r_i}} &= g^{(\tilde{\sigma}^1_{A_i}+\tilde{\rho}^1_{A_i}+\rho^1_i)+\ldots+(\tilde{\sigma}^m_{A_i}+\tilde{\rho}^m_{A_i}+\rho^m_i)} \\ &= g^{\tilde{\rho}^1_i+\ldots+\tilde{\rho}^m_i} \\ &= \pi(y^1_i)(\pi(u^1_i))^G\ldots\pi(y^m_i)(\pi(u^m_i))^G \qquad (21) \\ &= \pi(y_{r_i})(\pi(u_{r_i}))^G, \qquad (22) \end{aligned}$$

where the first and second lines follow our construction in Section 4.1, Formula 21 follows Lemma 1, and Formula 22 is a simplified form following Equations 15 and 16 (line 10 in Algorithm 3). Therefore, Formula 17 and Formula 18 are equal if their corresponding tuple items are equal as follows.

$$\begin{aligned} g^{\tilde{\sigma}_{r_i}} &= \tilde{y}_{r_i}(\tilde{u}_{r_i})^G, \\ g^{\tilde{\rho}_{r_i}} &= \pi(y_{r_i})(\pi(u_{r_i}))^G. \end{aligned}$$

The preceding equations are exactly enforced by Equations 11 and 12. This proves Theorem 1. □

5 Security

In this section, we analyze the security of SwiftParade against an attacker under the Dolev-Yao model [35]. The attacker can drop, deviate, alter, and inject packets (Section 3.2). All these attacks can be relatively straightforward to detect when no forging is involved.

Packet drop. The solution against packet-drop attacks usually leverages fault localization protocols to detect erroneous routers [36]. It is orthogonal to path validation [7].

Packet deviation. If a packet is directed to a router that is not supposed to be on any of the packet's permitted paths, its SessionID cannot match with a recorded one on the router (Section 4.3). This enables the router to easily detect the mis-forwarded packet.

Packet alteration. Simply altering packet data fails packet verification. For example, an altered payload leads to a DataHash that mis-matches with the one used for proof construction. Alteration of any other proof fields in the packet header also fails packet verification (Algorithm 3).

Packet injection. Without the attempt to forging proofs, the attacker may inject packets by replaying captured valid packets. It is also orthogonal to path validation for defending against such a replay attack. Common countermeasures verify TimeStamp freshness in packets [7] or record packet history on routers [37].

However, what further complicate security guarantee of path validation are various forging attacks. We next investigate three possible forging attacks against path validation in the literature. The analytical results show that SwiftParade can hardly be circumvented by these forged proofs.

5.1 Brute-force Attack

We start with analyzing the most straightforward brute-force attack. The length of path validation proofs renders a brute-force attack negligible [11], [12]. Specifically, a valid SwiftParade proof should be computed using correct values of all these fields—SessionID, Timestamp, DataHash, $\tilde{\sigma}_{r_i}$, $\tilde{\rho}_{r_i}$, $\tilde{u}_{r_i}$, and $\pi(u_{r_i})$. Our analysis in Section 6.1 shows that a 390-byte proof can enable SwiftParade to guarantee a sufficient security level. Then the probability of forging a valid proof via a brute-force attack is $2^{-(390\times 8)} = 2^{-3120}$. The attacker thus needs to forge a valid proof every $2^3120/2 = 2^3119$ tries on average. Such a large-scale traffic with invalid proofs can be easily detected and mitigated as distributed denial-of-service (DDoS) attacks [38], [39], countermeasures against which are orthogonal to path validation.

5.2 Selective-forging Attack

Due to the infeasibility of brute-force attacks, the attacker may seek to selectively forge key proof fields. This alternative is worthy of exploitation because some proof fields are computed using other fields. Then the attacker can take some less–path-sensitive fields (e.g., SessionID, Timestamp, and DataHash already in plaintexts) as is and then concentrate on only those fields requiring credentials for computation. According to Thoerem 1, the attacker can try to achieve so by selectively forging $\tilde{\sigma}_{r_i}$ and $\tilde{u}_{r_i}$. Let $\acute{\sigma}_{r_i}$ and $\acute{u}_{r_i}$ denote the respective forged fields.

Next, we show that the attacker can hardly forge selective proof fields in a polynomial time. Given a generator g, a composite number N, a public key y_i of router R_i, and a valid proof $\text{Proof}_{i-1} = (\tilde{\sigma}_{r_{i-1}}, \tilde{\rho}_{r_{i-1}})||\tilde{u}_{r_{i-1}}||\pi(u_{r_{i-1}})$ generated by its previous honest router R_{i-1}, we consider the following forged proof.

$$\acute{\text{Proof}}_i = (\acute{\sigma}_{r_i}, \acute{\rho}_{r_i})||\acute{u}_{r_i}||\acute{\pi}(u_{r_i}).$$

We now follow proof by contradiction. If $\acute{\sigma}_{r_i}$ and $\acute{u}_{r_i}$ can be forged in a polynomial time, $\acute{\rho}_{r_i}$ and $\acute{\pi}(u_{r_i})$ can be easily computed as $\acute{\rho}_{r_i} = \tilde{\rho}_{r_{i-1}} + \acute{\sigma}_{r_i}$ and $\acute{\pi}(u_{r_i}) = \pi(u_{r_{i-1}}) \times \acute{u}_{r_i}$, respectively. By Theorem 1, $\acute{\sigma}_{r_i}$ and $\acute{u}_{r_i}$ should satisfy the following condition:

$$g^{\acute{\sigma}_{r_i}} = \tilde{y}_{r_i}(\acute{u}_{r_i})^G. \tag{23}$$

Since $\acute{\sigma}_{r_i}$ should be incremented from the known parameter $\tilde{\sigma}_{r_{i-1}}$ (Equation 3), we denote $\acute{\sigma}_{r_i}$ as $\acute{\sigma}_{r_i} = \tilde{\sigma}_{r_{i-1}} + \alpha$ and derive the preceding condition as follows.

$$g^{\acute{\sigma}_{r_i}} = g^{\tilde{\sigma}_{r_{i-1}}+\alpha} = \tilde{y}_{r_{i-1}}(\tilde{u}_{r_{i-1}})^G g^\alpha. \tag{24}$$

Following Equation 23 and Equation 24, we further derive the subsequent equations for α to satisfy.

$$(\acute{u}_{r_i})^G = (y_{r_i})^{-1}(\tilde{u}_{r_{i-1}})^G g^\alpha. \tag{25}$$

$$\alpha = \log_g(\frac{\acute{u}_{r_i}}{\tilde{u}_{r_{i-1}}})^G y_{r_i}. \tag{26}$$

The attacker accordingly has two ways for proof forging. Neither of both ways is exploitable.

- By Equation 25, given an arbitrary forged value of $\acute{u}_{r_i}$, if α can be quickly solved, the attacker can therefore further forge $\acute{\sigma}_{r_i}$ as well as other fields. However, solving α in Equation 25 resembles a discrete logarithm problem (DLP) in Equation 26 that has no polynomial-time solution [40]. This renders $\acute{\sigma}_{r_i}$ and thus the entire proof hardly forgeable in a polynomial time.
- The attacker may turn to directly forge $\acute{\sigma}_{r_i}$. Then we can derive $\alpha = \acute{\sigma}_{r_i} - \tilde{\sigma}_{r_{i-1}}$ and $(\acute{u}_{r_i})^G = (y_{r_i})^{-1}(\tilde{u}_{r_{i-1}})^G g^\alpha$ by Equation 25. However, α and $\acute{u}_{r_i}$ should satisfy $\alpha = \sum_{j=1}^m \sigma_i^j = \sum_{j=1}^m (x_i^j + c_i^j H(h_i^j)G)$ (Section 4.1) and $\acute{u}_{r_i} = \tilde{u}_{r_{i-1}} g^{\sum_{j=1}^m c_i^j H(h_i^j)}$ (Section 4.4), respectively. To satisfy these requirements, the attacker needs to solve random value c_i and private key x_i given forged α and $\acute{u}_{r_i}$. This calls for DLP solutions again and prohibits the attacker from forging valid proofs in a polynomial time.

5.3 Quadratic-residue Attack

A recent quadratic-residue attack tries to circumvent path validation by exploiting a vulnerability of ElGamal encryption [31] used in proof `Construction` (Algorithm 2). Specifically, the exploited vulnerability exists if there exists a quadratic residue modulo N. An integer q is a quadratic residue modulo N if there exists an integer x such that $x^2 \equiv q \bmod N$. A possible q can be efficiently found if

TABLE 3: Validation-field size (in byte) of SwiftParade under an 80-bit security level.

Field	Size	Field	Size
Total	406	$\tilde{\sigma}_{r_i}$	63
DataHash	16	$\tilde{\rho}_{r_i}$	63
SessionID	4	$\tilde{u}_{r_i}$	128
Timestamp	4	$\pi(u_{r_i})$	128

N is either a prime number or can be easily factorized into several prime numbers [31]. Once a quadratic residue modulo N is found, it enables the attacker to forge a valid proof with a probability greater than $1/4$ [31].

To secure SwiftParade against the quadratic-residue attack, we construct N as a composite number of two primes ($p' \times p''$) instead of a single prime number (Section 4.2). Such a countermeasure is motivated by the fact that the attacker has to factorize a composite N for computing quadratic residues [31]. This resembles a prime factorization problem (PFP) that has no polynomial time solution [41]. For example, general number field sieve (GNFS)—known as the fastest algorithm for factoring a large composite number in number theory—holds a computation complexity of $\mathcal{O}(e^{1.9^3\sqrt{\lg N}^3\sqrt{(\lg\lg N)^2}})$ [42].

6 Evaluation

In this section, we implement and evaluate SwiftParade using DPDK [33] in comparison with Atlas [15]. We run SwiftParade on two 4-core server with Intel Xeon Platinum 8369B CPUs (3.5 GHz) and 8 GB memory. Performance evaluation focuses on path validation overhead in terms of communication and computation. Extensive results demonstrate that SwiftParade offers higher efficiency and applicability for multipath validation.

6.1 Validation-Field Size

We start with measuring the communication overhead of SwiftParade in terms of validation-field size. As shown in Table 3, SwiftParade introduces seven fixed-size validation fields to a packet header. The sizes of DataHash, SessionID, and Timestamp follow common setups in the literature. For the remaining four fields, their sizes are determined by the security level as follows. The dynamic rekeying technique suggested in [11] demonstrates that an 80-bit security level is sufficient. It does not necessarily enforce the conventional 128-bit security level because key setups in path validation may not have to be fixed. Once keys are re-established, accumulated attack efforts targeting at obsolete keys are immediately vanished. We next analyze the sizes of fields dependent on the security level.

- $\tilde{\sigma}_{r_i}$ is the summation of $\tilde{\sigma}_i$, while $\tilde{\sigma}_i$ is the aggregation result of σ_i. Let $\log_2 \sigma_i$ denote the length of σ_i. According to Equations 2, 3, and 5, the length of σ_i is bounded by $\log_2 c_i H(h) G + \log_2 i + \log_2 m$. Given that i denotes the index of an on-path router and m denotes the number of aggregated packets, they are upper bounded by the path length and queue size, respectively. We conservatively consider an extremely long path with 4,096 routers, each of which with a sufficiently large queue size of 4,096. The size of $\tilde{\sigma}_{r_i}$ can be estimated as $\log_2 c_i + \log_2 H(h) + \log_2 G + 24$ bits, where $\log_2 c_i$, $\log_2 H(h)$, and $\log_2 G$ are determined by the security level. Moreover, c_i represents a private key and both $H(h)$ and G are hash values. Given an 80-bit security level, the upper bound on the size of $\tilde{\sigma}_{r_i}$ is $(160 + 160 + 160 + 24)$ bits = 63 bytes.
- $\tilde{\rho}_{r_i}$ shares the same computation logic as $\tilde{\sigma}_{r_i}$. Therefore, its size is also 63 bytes given the same security level.
- $\tilde{u}_{r_i}$ and $\pi(u_{r_i})$ are both computed using the modulo base N. Their lengths are thus upper bounded by the length of N. To satisfy an 80-bit security level, N is 1,024-bit long [43]. Therefore, the lengths of both $\tilde{u}_{r_i}$ and $\pi(u_{r_i})$ are 1,024 bits (i.e., 128 bytes).

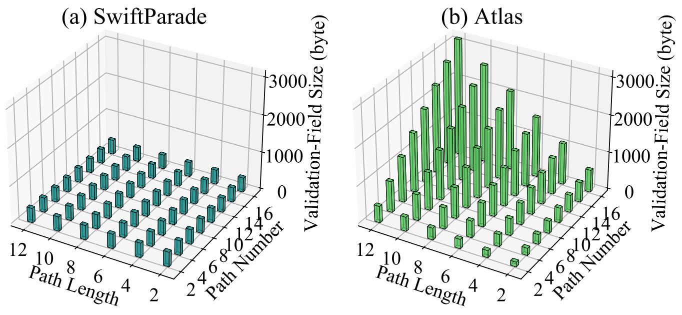

Fig. 3: Proof size of SwiftParade and Atlas [15] with varying path number and path length.

In summary, the length of extra header fields by SwiftParade is 406 bytes under an 80-bit security level (Table 3). Note that it is constant regardless of path number and path length.

In contrast, the communication overhead of Atlas increases with both path number and path length. The size of Atlas validation fields is estimated as [15]:

$$16(m(n-1)-(k-1)) + 18(m+k) + 36 \text{ (bytes)},$$

where m is path number, n is path length, and k is the number of diverging routers in the multipath topology. Such a high complexity of $\mathcal{O}(mn)$ significantly increases the communication overhead of Atlas in complex topologies.

Figure 3 compares SwiftParade with Atlas in terms of communication overhead under various topologies. SwiftParade associates with a constant overhead while Atlas leads to a higher overhead as path number or path length increases. SwiftParade starts to outperform Atlas when the topology becomes complex. Consider the topology in Figure 2 for example—4 paths with 6 routers on each path and 3 diverging routers. Atlas introduces 418 bytes of validation fields while SwiftParade requires only 406 bytes. Furthermore, Atlas may become ineffective when topology complexity becomes moderately high (e.g., in a data center with 1,280 routers following the fat-tree topology [44]). As shown in Figure 3, given a multipath topology with path number of 8 and path length of 12, Atlas necessitates 1,610 bytes of validation fields that already exceed the 1,500-byte MTU limit. We report only Atlas statistics without MTU limit violation for practicality in what follows.

6.2 Proof Construction

Figure 4 reports the comparison of proof construction time per packet on the source of SwiftParade and Atlas [15]. We

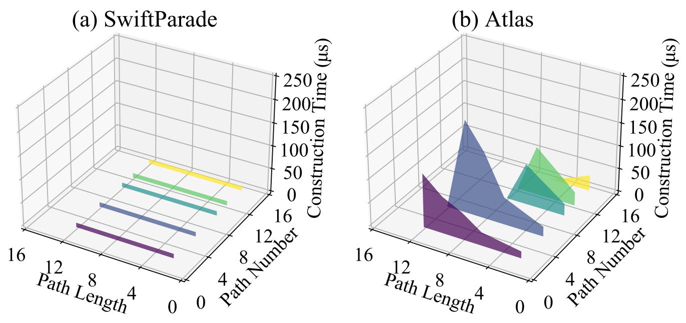

Fig. 4: Construction time per packet of SwiftParade and Atlas with varying path number and path length.

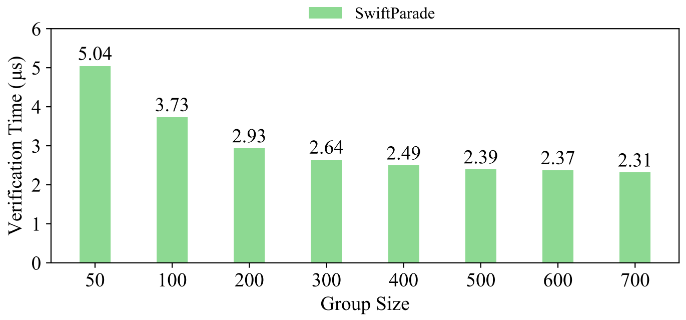

Fig. 5: Verification time per packet of SwiftParade with varying group size.

measure the average construction time over 10,000 packets. SwiftParade can construct proofs extremely fast because it only computes the identification of the source rather than all the on-path routers into the proof. This offers a constant construction time of 9.13 μs per packet (Figure 4(a)). However, Atlas needs to calculate proofs for every router in the multipath topology (Section 2.2). This results into an $\mathcal{O}(mn)$ computation complexity, where m denotes path number and n denotes path length. The proof construction time of Atlas thus increases with path number and path length (Figure 4(b)). For example, given a 6-path multipath topology, the proof construction time of Atlas is 58.95 μs and 200.19 μs when path length is 6 and 12, respectively. When path number increases to 10, the proof construction time per packet by Atlas increases by 48.96% given path length of 6. Longer paths make Atlas inapplicable to multipath validation due to MTU violation.

6.3 SwiftParade Verification with Varying Group size

Prior to comparing the proof verification time of SwiftParade and Atlas, we first evaluate how group size affects SwiftParade in terms of verification speed. Group size quantifies the number of packets that SwiftParade can process in one round of aggregate validation. Figure 5 shows the average verification time per packet with varying group size. The more packets are aggregated into a group, the less verification time is needed per packet in the group. When group size increases from 50 to 400, the verification time decreases from 5.04 μs to 2.49 μs, offering a 50.60% speedup. Note that both path length and path number do not affect the efficiency of SwiftParade as the only related parameter involved in aggregate validation is the packet number (Algorithm 3).

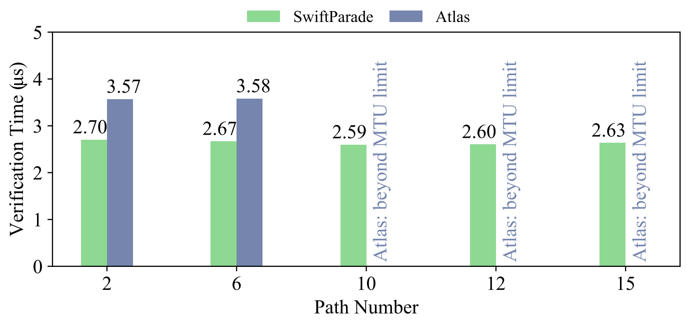

Fig. 6: Verification time per packet of SwiftParade and Atlas with 10-hop paths and varying path number.

6.4 Proof Verification with Varying Path Number

We now measure the verification time of SwiftParade in comparison with that of Atlas. We start with the evaluation configuration with a fixed path length of 10 and varying path number. We use each path to transfer 150 packets to emulate burst arrivals (Section 2.1). According to the evaluation results with varying group size in Figure 5, we set the group size as 300 as verification speedup tends to cease upon larger group sizes. Figure 6 compares the average verification time per packet under varying path numbers. Once path length is fixed, both SwiftParade and Atlas deliver relatively constant verification time per packet regardless of path number. SwiftParade outperforms Atlas with a 1.35× speedup.

6.5 Proof Verification with Varying Path Length

We continue to measure the verification time with varying path length. The number of multi-paths is set as 6 (Figure 6). Similarly, we send 150 packets along each path to emulate burst arrivals and group 300 packets on routers for aggregate validation. Figure 7 reports the average verification time per packet by SwiftParade and Atlas. SwiftParade remains a constant validation speed about 2.12 μs as its computation only relates to the inputs from the previous-hop routers (Section 4.4). In contrast, the verification time of Atlas increases with path length because the PVF field it re-computes requires inputs from all the upstream routers. Furthermore, the number of Atlas proof fields increases with path length. This also increases the time for routers to locate the specific proof for verification. For example, Atlas takes 1.94 μs for verifying a packet given a path length of 6. The verification time increases by 82.59% when path length increases to 12. In summary, SwiftParade starts outperforming Atlas as path length exceeds 6 and the advantage expands with path length. For example, SwiftParade has a 1.42× speedup in comparison with Atlas when path length is 10. The speedup increases to 1.67× when path length is up to 12.

6.6 Proof Processing Time

Finally, we measure the total proof processing time. The constitution of processing time varies across different routers. According to Algorithm 2, the source router process proofs via only `Construction`. The intermediate routers first verify proofs via `Verification` (Algorithm 3) and then update proofs via `Construction` (Algorithm 2) for validated packets. The destination router only needs to verify

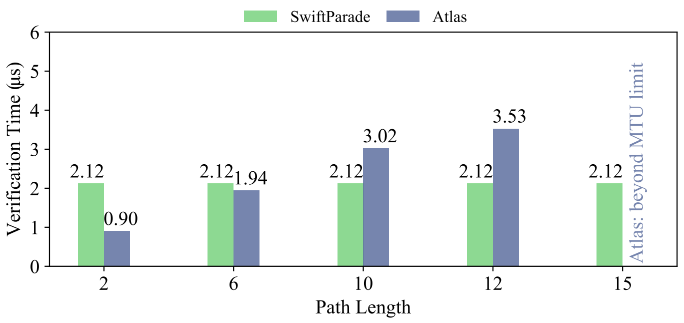

Fig. 7: Verification time per packet of SwiftParade and Atlas with 6 variable-length paths.

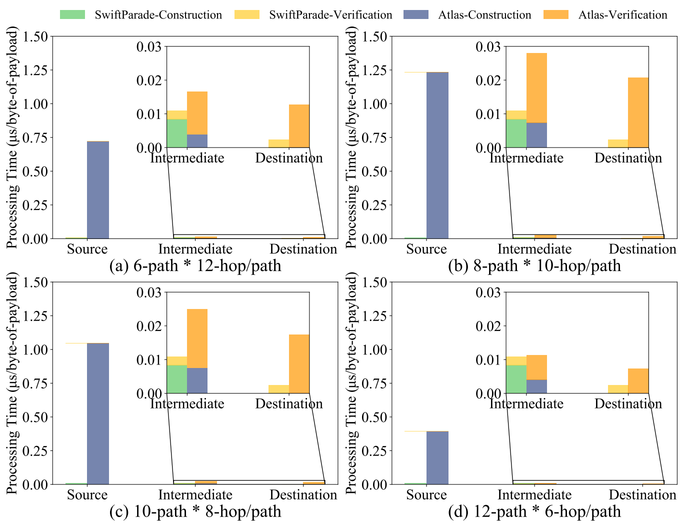

Fig. 8: Proof processing time per byte of payload of SwiftParade and Atlas with varying path number and path length. For each chosen path number, path length is set as the largest one that leads to no MTU violation for Atlas.

proofs via `Verification` (Algorithm 3). However, we find that it is unfair for SwiftParade to simply estimate the average proof processing time per packet. This is because SwiftParade introduce shorter proofs than Atlas does. Given the same number of packets, SwiftParade likely carries way more payloads. Therefore, we turn to evaluate the average processing time per byte of payload and reports the comparison results of SwiftParade and Atlas in Figure 8. The routing topology is set with varying paths and hops per path. Given a path number, we choose the maximum path length that does no violate the MTU limit for Atlas. The group size is set as 300 following the preceding experiments.

Source router. The proof processing time of SwiftParade on the source is constant as 0.0083 μs/byte-of-payload. In contrast, the processing time of Atlas depends on routing topology specifics (Section 6.2). For example, with 12 paths and 6 hops per path (Figure 8 (d)), the payload per packet supported by Atlas is up to $1{,}500 - 1{,}234 = 266$ bytes, where 1,234 corresponds to the length of Atlas proofs. In this case, the processing time of Atlas is 0.39 μs/byte-of-payload. When path number is 8 and each path has 10 hops (Figure 8 (b)), the processing time of Atlas is 1.23 μs/byte-of-payload, being 148.19× slower than that of SwiftParade.

Intermediate router. SwiftParade constinues to deliver constant-time processing on intermediate routers simply because every router performs the same computation (`Verification` in Algorithm 3 and `Construction` in Algorithm 2). It takes 0.0025 μs/byte-of-payload to verify proofs and 0.0083 μs/byte-of-payload to update proofs. Thus, the total proof processing time of SwiftParade on an intermediate router is 0.0110 μs/byte-of-payload. As for Atlas, its verification time is determined by path length (Section 6.5). The update time is as constant as 1.07 μs. Besides, since proof size of Atlas is not constant (Section 6.1), the payload size varies with path number and path length that decide proof size. Specifically, the proof processing time of Atlas is 0.0073 (verification) + 0.0040 (construction) = 0.0113 μs/byte-of-payload with 12 paths and 6 hops each (Figure 8 (d)). When the routing topology features with 6 paths and 12 hops each (Figure 8 (a)) and with 8 paths and 10 hop each (Figure 8 (b)), the processing time of Atlas is 0.0164 μs/byte-of-payload and 0.0280 μs/byte-of-payload, respectively. Respective levels of slowdown in comparison with SwiftParade are 32.93% and 60.71%.

Destination router. The proof processing time of SwiftParade on the destination is also faster than that of Atlas. The processing time of Atlas ranges from 0.0073 μs/byte-of-payload (Figure 8 (d)) to 0.0207 μs/byte-of-payload (Figure 8 (b)). In contrast, SwiftParade constantly takes 0.0025 μs/byte-of-payload, being 2.92×~8.28× faster than Atlas.

7 Related Work

In this section, we review related solutions and underline how our SwiftParade contributes atop them toward anti-burst multipath validation. Existing path validation schemes can be divided into two categories according to their application scenarios—single-path validation and multipath validation. Single-path validation enforces a strict binding between a packet and a specific forwarding path. Multipath validation allows packet forwarding along any one of a set of designated paths.

7.1 Single-path Validation

Single-path validation solutions needs to compute proofs using a priori knowledge about the exactly single specified path for a packet. They thus cannot apply to networks with parallel and dynamic multipath routing, unsatisfying the multipath scenario listed in Table 1. We next review typical single-path validation solutions.

Early path validation solutions tend to trade off security for efficiency. As the pioneer path validation solution, ICING [6] enforces the strongest security assumption that each node needs to verify all its upstream nodes and compute proofs for all its downstream nodes. Such heavy computation limits its efficiency. To improve efficiency, OPT [7] assumes a trusted source that can pre-compute various proof fields for on-path routers. Routers then need to only verify and update proofs. Following similar primitives, EPIC [12] further simplifies the computation and decreases proof size. All these solutions adopt symmetric encryption and associate each router with a specific proof, resulting in an $\mathcal{O}(n)$ storage complexity. This renders them lack of the property of constant-size proof in Table 1.

Toward constructing constant-size proofs, Atomos [11] uses asymmetric encryption instead. It aggregates proofs of each router into one single proof field. However, due to the relatively slow asymmetric encryption, Atomos is more suitable for networks with long forwarding paths.

When security guarantee needs not be limited to every path segment per packet, there have been more alternative solutions aiming at high efficiency. For example, OSV [8], [9] replaces the traditional cryptographic computation such as message authentication code (MAC) with a Hadamard matrix. The simplicity of matrix computation enables OSV to be much faster than those cryptographic solutions. PPV [10] proposes a probabilistic validation scheme that no longer requires a packet be verified by only one path segment as well as by the destination. Similarly, MASK [14] empowers the source to designate an intermediate router to generate a proof for the specified packet. Then the packet embedded with the proof is only verified at the destination.

All the preceding solutions target at networks with static paths. However, once the routing policy changes, routers have to drop packets in forwarding since their proofs still correspond to the on-path routers specified by the obsolete routing policy. PSVM [13] aims to adapt to dynamic routing by introducing a trusted agent called Credible Guarantee Agent (CGA). CGA informs the corresponding router with the new routing decision and issues the newly precomputed proofs to replace the carried proofs in the packet header. This strategy relies on a trusted authority, which weakens the security assumption. Besides, there is a delay in the arrival of the newly-calculated proof to the router. This renders PSVM less suitable for complex and dynamic networks.

7.2 Multipath Validation

Recent studies start to focus on multipath validation that allows a packet to switch among multiple allowed paths during transmission [15], [16]. As the first attempt toward multipath validation, Atlas [15] proposes a hierarchical validation technique to compress proofs of multiple paths. Paths are split into segments. Proofs for overlapping segments on multiple paths are computed only once. The follow-up privacy-preserving solution—VALNET [16]—leverages chameleon hash [45] to hide hop indices of routers on a certain path. This, however, involves additional computation and communication overhead. We in this paper present SwiftParade toward a more efficient multi-path validation solution in comparison with Atlas. We consider exploring further improvements regarding privacy protection as VALNET for future work.

8 Conclusion

We have presented SwiftParade as the first attempt toward anti-burst multipath validation. Existing solutions enforce packet-wise validation; their validation overhead is imposed equally on every single packet. This renders them less applicable to multipath routing where bursty traffic becomes a norm. Such bursty traffic arises from simultaneous incoming packets along multiple forwarding paths toward the same router. Without an effective way to improve validation efficiency, bursty traffic tends to overload validation resources and leads to packet losses. We explore an aggregate validation technique to fundamentally improve validation efficiency. Specifically, it simultaneously validates a group of packets from different paths. The entire group of packets can be validated as long as each packet follows any regulated path. Then the validation overhead is amortized across packets of the same group instead of being equally imposed on each packet. To implement the aggregate validation technique, we develop a noncommutative homomorphic asymmetric encryption scheme. Extensive theoretical and evaluation results demonstrate that SwiftParade outperforms the state-of-the-art multipath validation solution.

Acknowledgments

We would like to sincerely thank the Editors and Reviewers of IEEE Transactions on Dependable and Secure Computing for your review efforts and helpful feedback. We also wish you health and safety during the pandemic. 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. 2022YFB3104800 and Grant No. 2020AAA0107705, National Natural Science Foundation of China under Grant No. 62032021, Zhejiang Key R&D Plan under Grant No. 2019C03133, Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang under Grant No. 2018R01005, Alibaba-Zhejiang University Joint Institute of Frontier Technologies, and Research Institute of Cyberspace Governance in Zhejiang University.

References

[1] Y. Sun, A. Edmundson, L. Vanbever, O. Li, J. Rexford, M. Chiang, and P. Mittal, "Raptor: Routing attacks on privacy in tor," in *USENIX Security Symposium*, 2015, pp. 271–286.

[2] H. Birge-Lee, Y. Sun, A. Edmundson, J. Rexford, and P. Mittal, "Bamboozling certificate authorities with {BGP}," in *USENIX Security Symposium*, 2018, pp. 833–849.

[3] L. D. Amini, A. Shaikh, and H. G. Schulzrinne, "Issues with inferring internet topological attributes," in *Internet Performance and Control of Network Systems III*, vol. 4865. International Society for Optics and Photonics, 2002, pp. 80–90.

[4] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira, "Avoiding traceroute anomalies with paris traceroute," in *SIGCOMM*, 2006, pp. 153–158.

[5] K. Bu, A. Laird, Y. Yang, L. Cheng, J. Luo, Y. Li, and K. Ren, "Unveiling the mystery of internet packet forwarding: A survey of network path validation," *ACM Computing Surveys*, vol. 53, no. 5, pp. 1–34, 2020.

[6] J. Naous, M. Walfish, A. Nicolosi, D. Mazières, M. Miller, and A. Seehra, "Verifying and enforcing network paths with icing," in *CoNEXT*, 2011, pp. 1–12.

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

[8] H. Cai and T. Wolf, "Source authentication and path validation with orthogonal network capabilities," in *INFOCOM WKSHPS*, 2015.

[9] ——, "Source authentication and path validation in networks using orthogonal sequences," in *ICCCN*, 2016, pp. 1–10.

[10] B. Wu, K. Xu, Q. Li, Z. Liu, Y.-C. Hu, M. J. Reed, M. Shenk, and F. Yang, "Enabling efficient source and path verification via probabilistic packet marking," in *IWQoS*, 2018, pp. 1–10.

[11] A. He, K. Bu, Y. Li, E. Chida, Q. Gu, and K. Ren, "Atomos: Constant-size path validation proof," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 3832–3847, 2020.

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

[13] F. Yang, K. Xu, Q. Li, R. Lu, B. Wu, T. Zhang, Y. Zhao, and M. Shen, "I know if the journey changes: Flexible source and path validation," in *IWQoS*, 2020, pp. 1–6.

[14] S. Fu, K. Xu, Q. Li, X. Wang, S. Yao, Y. Guo, and X. Du, "Mask: Practical source and path verification based on multi-as-key," in *IWQoS*, 2021, pp. 1–10.

[15] L. Ma, K. Bu, N. Wu, T. Luo, and K. Ren, "Atlas: A first step toward multipath validation," *Computer Networks*, vol. 173, p. 107224, 2020.

[16] B. Sengupta, "Valnet: Privacy-preserving multi-path validation," *Computer Networks*, vol. 204, p. 108695, 2022.

[17] X. Yang, D. Clark, and A. W. Berger, "Nira: a new inter-domain routing architecture," *IEEE/ACM Transactions on Networking*, vol. 15, no. 4, pp. 775–788, 2007.

[18] T. Anderson, K. Birman, R. Broberg, M. Caesar, D. Comer, C. Cotton, M. J. Freedman, A. Haeberlen, Z. G. Ives, A. Krishnamurthy *et al.*, "The nebula future internet architecture," in *The Future Internet Assembly*, 2013, pp. 16–26.

[19] X. Zhang, H.-C. Hsiao, G. Hasker, H. Chan, A. Perrig, and D. G. Andersen, "Scion: Scalability, control, and isolation on next-generation networks," in *S&P*. IEEE, 2011, pp. 212–227.

[20] D. Barrera, L. Chuat, A. Perrig, R. M. Reischuk, and P. Szalachowski, "The scion internet architecture," *Communications of the ACM*, 2017.

[21] J. Kwon, J. A. García-Pardo, M. Legner, F. Wirz, M. Frei, D. Hausheer, and A. Perrig, "Scionlab: A next-generation internet testbed," in *ICNP*, 2020, pp. 1–12.

[22] IETF, "Multipath tcp (mptcp)," 2020. [Online]. Available: https://datatracker.ietf.org/wg/mptcp/documents/

[23] I. Cidon, R. Rom, and Y. Shavitt, "Analysis of multi-path routing," *IEEE/ACM transactions on Networking*, vol. 7, no. 6, pp. 885–896, 1999.

[24] C. Hopps, "Analysis of an equal-cost multi-path algorithm," Tech. Rep., 2000.

[25] M. Besta, M. Schneider, M. Konieczny, K. Cynk, E. Henriksson, S. Di Girolamo, A. Singla, and T. Hoefler, "Fatpaths: Routing in supercomputers and data centers when shortest paths fall short," in *SC*, 2020, pp. 1–18.

[26] S. Kandula, D. Katabi, B. Davie, and A. Charny, "Walking the tightrope: Responsive yet stable traffic engineering," in *SIGCOMM*, 2005, pp. 253–264.

[27] H. Wang, H. Xie, L. Qiu, Y. R. Yang, Y. Zhang, and A. Greenberg, "Cope: Traffic engineering in dynamic networks," in *SIGCOMM*, 2006, pp. 99–110.

[28] S. Kandula, D. Katabi, S. Sinha, and A. Berger, "Dynamic load balancing without packet reordering," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 2, pp. 51–62, 2007.

[29] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall, "Let it flow: Resilient asymmetric load balancing with flowlet switching," in *NSDI*, 2017, pp. 407–420.

[30] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker, "On the characteristics and origins of internet flow rates," in *SIGCOMM*, 2002, pp. 309–322.

[31] Y. Wu, C. Jiang, C. Xu, and K. Chen, "Security analysis of a path validation scheme with constant-size proof," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 4246–4248, 2021.

[32] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp. 469–472, 1985.

[33] DPDK: Data Plane Development Kit,. [Online]. Available: http://dpdk.org/

[34] M. M. Hamdi, S. A. Rashid, M. Ismail, M. A. Altahrawi, M. F. Mansor, and M. K. AbuFoul, "Performance evaluation of active queue management algorithms in large network," in *ISTT*, 2018, pp. 1–6.

[35] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–208, 1983.

[36] C. Basescu, Y.-H. Lin, H. Zhang, and A. Perrig, "High-speed inter-domain fault localization," in *IEEE S&P*, 2016, pp. 859–877.

[37] T. Lee, C. Pappas, A. Perrig, V. Gligor, and Y.-C. Hu, "The case for in-network replay suppression," in *AsiaCCS*, 2017, pp. 862–873.

[38] J. Mirkovic, G. Prier, and P. Reiher, "Attacking DDoS at the source," in *ICNP*, 2002, pp. 312–321.

[39] S. T. Zargar, J. Joshi, and D. Tipper, "A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 4, pp. 2046–2069, 2013.

[40] W. Diffie and M. E. Hellman, "New directions in cryptography," in *Secure Communications and Asymmetric Cryptosystems*. Routledge, 2019, pp. 143–180.

[41] R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public key cryptosystems," in *Secure Communications and Asymmetric Cryptosystems*. Routledge, 2019, pp. 217–239.

[42] A. K. Lenstra, H. W. Lenstra, M. S. Manasse, and J. M. Pollard, "The number field sieve," in *The Development of the Number Field Sieve*. Springer, 1993, pp. 11–42.

[43] A. K. Lenstra and E. R. Verheul, "Selecting cryptographic key sizes," *Journal of Cryptology*, vol. 14, no. 4, pp. 255–293, 2001.

[44] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4, pp. 63–74, 2008.

[45] H. Krawczyk and T. Rabin, "Chameleon hashing and signatures," *Cryptology ePrint Archive*, 1998.