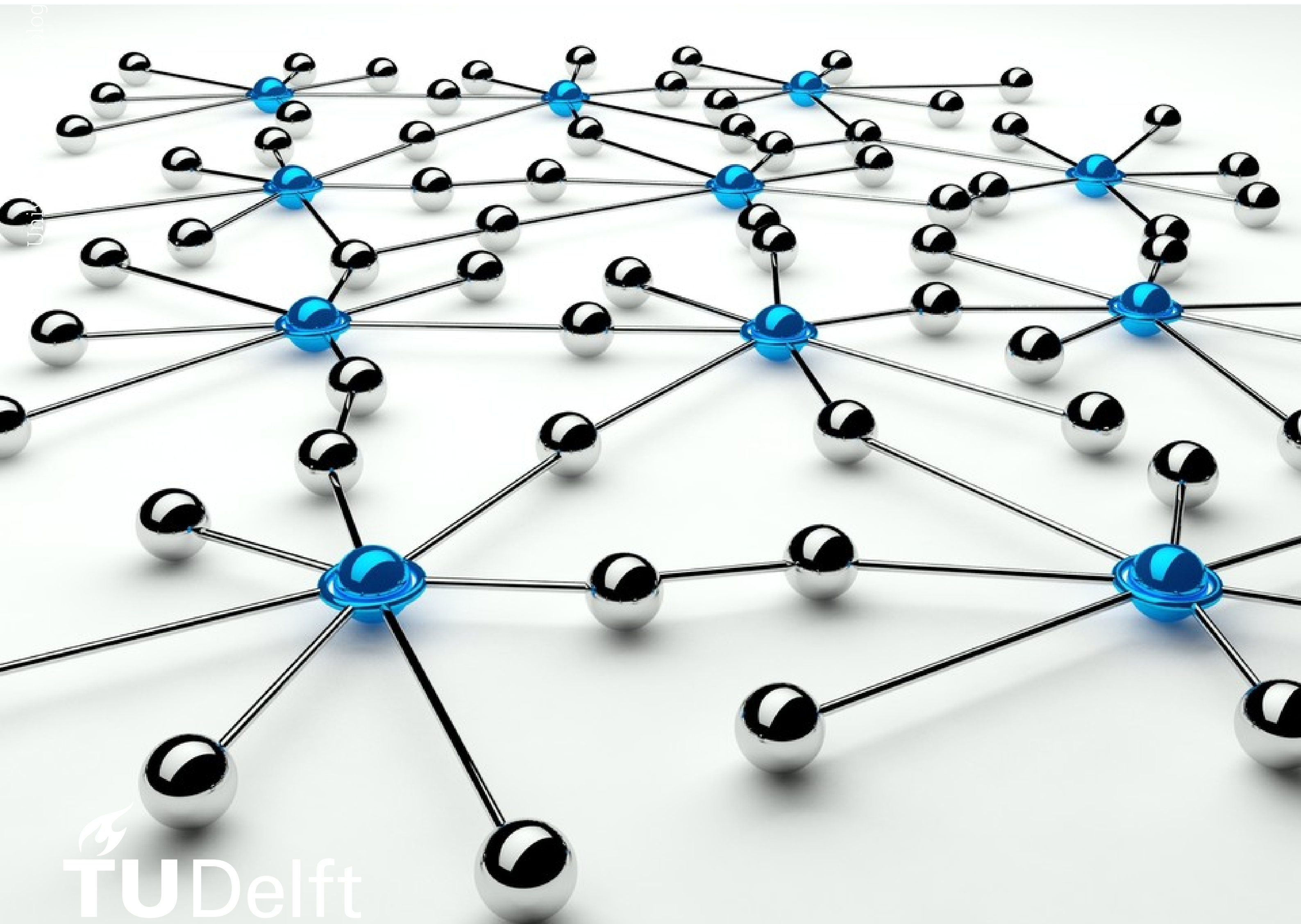
Path Verification for Controllable Routing

Path Verification with Per-Hop Key Exchange Using Programmable Data Planes

MSc Thesis — Computer Science

Callum Holland



Path Verification for Controllable Routing

Path Verification with Per-Hop Key Exchange Using Programmable Data Planes

by

Callum Holland

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on 24th of June 2026.

Student number:	4462211	
Project duration:	May 2025 – June 2026	
Thesis committee:	ir. Adrian Zapletal,	TU Delft, supervisor
	Dr. F. Kuipers,	TU Delft
	Dr. A. Voulimeneas,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.
The source code is available at `https://github.com/CallumH2410/Hermes`.

TU Delft

Preface

After a long time of being a student, the time has finally come to an end to say goodbye to what has been an extremely fun chapter of my life. I always dreaded this moment, which is probably why (sub)consciously procrastinated giving the final hurdle a go. This wasn't because I didn't ever want to receive the piece of paper I've paid many euros on tuition fees on, but because I didn't want to face the harsh reality that I now have to leave the most exciting time of my life behind. I would like to thank all the people I have met along the way who made this an unforgettable experience.

I would also like to thank my work for being extremely flexible with my working hours during this period.

To my family, although it has taken a long time and they never failed to remind me whenever I saw or spoke to them, thank you for your continued support throughout this journey.

As final hurdles go, this was a very enjoyable process. The free rein I was given at the start of the project allowed me to be as creative as possible. Even though my first idea came to a harsh and abrupt end, the new turn it took was also fun developing and I can be proud of the end product.

I would like to thank Dr. Fernando Kuipers for the time he spent as Chair of my Thesis, including the meetings and administrative support throughout this project. I would also like to thank Gabe for helping me with the implementation on the physical Tofino switches and for showing interest in my project whenever I stopped by Adrian's office for our weekly meetings. Finally, I would like to thank my daily supervisor, Adrian Zapletal, who always had time to provide input, feedback, and guidance whenever I needed it. I am deeply appreciative of the amount of time, effort, and interest he invested throughout the duration of this project.

Finally, a massive thank you to my girlfriend, Liss. She has supported me throughout this journey, listened patiently whenever my stress levels rose, and always knew how to offer wise advice when I needed it most.

One quote has always stuck with me:

"I wish there was a way to know you're in the good old days before you've actually left them." — Andy Bernard

I'm happy to say that I knew I was in the good old days.

Abstract

Modern Internet routing provides little visibility or control to end users over the paths their packets take through the network. This thesis investigates whether it is feasible to design and implement a routing verification protocol that (i) lets a sender steer packets through a user-defined set of trusted switches and/or networks, and (ii) cryptographically proves, after transmission, that the packet traversed exactly the intended path and no other.

The proposed system, called *Hermes*, combines programmable data-plane technology (P4) with an in-band accumulator-based path attestation scheme. Each switch on the authorised path holds a private key derived through an optical Diffie–Hellman key exchange with the Hermes verification server. As a probe packet traverses the path, each switch applies one of eight defined bitwise operations using its current key to update a 32-bit accumulator embedded in the packet header. At the receiver, the accumulated value is forwarded to Hermes, which independently replays the computation and verifies correctness.

To resist eavesdroppers who might deduce per-switch keys by observing accumulator differences between a switch's input and output, the system uses a large pool of keys per switch. Keys are consumed sequentially and then reused in an order driven by a 32-bit Galois Linear Feedback Shift Register (LFSR), ensuring the key-use sequence is non-repeating and unpredictable. Hermes counters replay attacks by embedding per-packet sequence numbers, millisecond-resolution timestamps, and random nonces in the packet header.

Contents

Preface i

Summary ii

Nomenclature vii

1 Introduction 1

2 Background 3
2.1 Networking and Routing . . . 3
2.1.1 Autonomous Systems and the Internet's Trust Model . . . 3
2.1.2 Routing Protocols . . . 4
2.1.3 Source Routing and Segment Routing . . . 5
2.1.4 The Path-Visibility Problem . . . 5
2.2 Programmable Data Planes and P4 . . . 5
2.3 Cryptographic Foundations . . . 7
2.3.1 Symmetric Cryptography . . . 7
2.3.2 Asymmetric Cryptography . . . 7
2.3.3 Classical Diffie–Hellman Key Exchange . . . 7
2.3.4 Optical Diffie–Hellman (XOR-based Variant) . . . 7
2.3.5 Linear Feedback Shift Registers . . . 8
2.3.6 Message Authentication Codes . . . 8
2.3.7 Replay Attacks and Defences . . . 8
2.3.8 Transport Layer Security and Mutual TLS . . . 9
2.4 Threat Model . . . 9

3 Related Work 11
3.1 Routing Control and Trust Architectures . . . 11
3.1.1 Responsible Internet . . . 11
3.1.2 User Controlled Internet Protocol (UCIP) . . . 11
3.1.3 Mutually Controlled Routing (MCR) . . . 12
3.1.4 Pathlet Routing . . . 12
3.1.5 Outsourcing Routing Control via SDN . . . 12
3.1.6 SCION . . . 13
3.2 Packet-Level Path Verification . . . 13
3.2.1 ICING . . . 13
3.2.2 Lightweight Source Authentication and Path Validation (L-SPV) . . . 14
3.2.3 Proof-of-Provenance . . . 14
3.2.4 PFA-INT . . . 15
3.2.5 InvisiFlow . . . 15
3.3 Security in Programmable Data Planes . . . 16
3.3.1 In-band Network Telemetry (INT) . . . 16
3.3.2 dh-aes-p4 . . . 16
3.3.3 OAT: Attested Execution in the Data Plane . . . 17
3.3.4 P4-IPsec and P4-MACsec . . . 17
3.4 Performance-Driven Routing . . . 17
3.5 Summary and Positioning . . . 18

4 Hermes Design 19
4.1 Design Goals and Requirements . . . 19
4.2 Overview . . . 20

4.2.1 Actors and Components . . . 21
4.2.2 Operational Phases . . . 21
4.2.3 Per-Packet Verification Workflow . . . 22
4.3 Key Provisioning: Optical Diffie–Hellman . . . 22
4.4 Accumulator Update Rule . . . 23
4.4.1 Operations . . . 23
4.4.2 Rotation Operation . . . 23
4.4.3 Security Rationale . . . 24
4.5 Key Lifecycle . . . 24
4.6 Replay Protection . . . 24
4.7 Packet Header Format . . . 25
4.7.1 Message Types . . . 25
4.7.2 Data Packet Header . . . 25
4.7.3 Digest to the Control Plane . . . 26

5 Security Analysis 27
5.1 Security Goals and Threat Model . . . 27
5.2 Forgery Resistance . . . 28
5.3 Why Key Recovery Does Not Enable Hijacking . . . 29
5.4 Replay Resistance . . . 31
5.5 Conclusion . . . 31

6 Evaluation Methodology and Results 32
6.1 Research Questions . . . 32
6.2 Experimental Setup . . . 32
6.3 Experiment 1: Path-Deviation Detection . . . 34
6.4 Experiment 2: DH Key-Exchange Latency . . . 34
6.5 Experiment 3: End-to-End Verification Latency . . . 35
6.6 Experiment 4: Data-Plane Forwarding Throughput . . . 35
6.7 Experiment 5: Key Rotation Overhead . . . 36
6.8 Experiment 6: Header Overhead . . . 37
6.9 Experiment 7: Flow Completion Time . . . 38
6.10 Summary . . . 39

7 Discussion 41
7.1 Interpretation of the Results . . . 41
7.2 Security Trade-offs . . . 41
7.3 Scope and Generalisability . . . 42
7.4 Toward Multi-Domain Hermes Deployments . . . 42
7.5 Limitations . . . 43
7.6 Future Work . . . 43
7.7 Closing Perspective . . . 44

8 Conclusion 45

References 47

A P4 Header and Digest Definitions 49
A.1 Header Types . . . 49
A.2 Digest Type . . . 49
A.3 Opcode Definitions . . . 49

B Hermes Server Wire Protocol 51
B.1 DH Initialisation (`DH_INIT` / `DH_RESP`) . . . 51
B.2 Key Regeneration (`DH_KEY` / `DH_KEY_RESP`) . . . 51
B.3 Verification (`VERIFY` / `RESULT`) . . . 51

List of Figures

4.1 High-level Hermes architecture, illustrated for an example path of length $L = 4$ (generically $s_1, \ldots, s_L$, with sender h_1 and receiver h_2). 20

5.1 Number of (op, k) hypotheses consistent with one observation (≥ 20 clipped into the rightmost bar). The mixture averages to a single-guess forgery success of approximately 0.125. 29
5.2 Schedule and array-ordering recovery in a weakened 16-bit-LFSR model with key values already granted to the adversary. The LFSR seed itself falls quickly once the model is reduced, but full next-hop prediction (the hijack-ready capability) is gated by covering *both* independently shuffled secret arrays, and only approaches certainty after ~2,000–3,000 observed flows at one switch. At the real 32-bit seed a 2^{32} search is required in addition. 30

6.1 DH key-exchange RTT versus key count on `sw1` (30 runs each): median bar, min–max whisker, and p95 marker. 35
6.2 End-to-end verification latency by offered probe rate, split into digest latency and server-side verify latency. 36
6.3 Plain-forwarding baseline versus Hermes verification latency. The baseline forwarding RTT has median 0.095 ms, while the Hermes end-to-end verification median is approximately 5.88 ms. 36
6.4 Offered forwarding throughput over 10 s windows, Hermes versus the plain-forwarding baseline. Packet rate is flat for both, indicating a software-generator bottleneck. 37
6.5 Per-packet key-rotation overhead relative to a no-rotation baseline. Sequential and LFSR advances are pipeline-only; full renegotiation is control-plane dominated. 37
6.6 Per-packet header overhead versus path length L. Hermes adds a fixed 28 bytes regardless of L ($O(1)$), while ICING and INT add overhead that grows linearly with L ($O(L)$); shaded bands show the per-hop cost ranges given in the text. 38
6.7 Flow completion time as receiver span versus flow size, p50 and p99. Completion scales linearly with size above approximately 10 kB; small flows sit at the timing floor. 40
6.8 Flow completion time CDF as receiver span over multi-packet flows. The heavy tail corresponds to large flows. 40

List of Tables

3.1 Comparison of path verification and routing control mechanisms. 18

4.1 Accumulator update operations and their 3-bit opcodes. 23
4.2 Hermes message types. 25
4.3 Fields of the Hermes data header. 25
4.4 Fields of the Hermes digest format. The final hop populates all fields, including the verification-relevant accumulator and anti-replay fields; any switch signalling local key-pool exhaustion sets `trigger_dh` and `hop_idx`. 26

5.1 Security properties of Hermes against the defined threat model. 28
5.2 Candidate keys implied by one observation, per opcode hypothesis. 28

6.1 Hardware and software environment. 33
6.2 Path-deviation detection (30 trials each). All cases classified correctly. 34
6.3 Per-packet header overhead introduced by Hermes. 38
6.4 Workload and capture summary (3,000 flows, single run). 39

Nomenclature

Abbreviations

Abbreviation	Definition
AEAD	Authenticated Encryption with Associated Data
AS	Autonomous System
ASN	Autonomous System Number
BF-RT	Barefoot Runtime, the control-plane API used to program Intel Tofino switches
BGP	Border Gateway Protocol
BMv2	Behavioural Model version 2 (software P4 switch)
CA	Certificate Authority
DH	Diffie–Hellman
FHE	Fully Homomorphic Encryption
gRPC	Google Remote Procedure Call framework
INT	In-band Network Telemetry
ISD	Isolation Domain (SCION)
LFSR	Linear Feedback Shift Register
LSA	Link-State Advertisement
MAC	Message Authentication Code
mTLS	Mutual Transport Layer Security
OSPF	Open Shortest Path First
P4	Programming Protocol-Independent Packet Processors
PCFS	Packet-Carried Forwarding State
PHE	Partially Homomorphic Encryption
PRF	Pseudo-Random Function
PVF	Path Validation Field
RIP	Routing Information Protocol
RTT	Round-Trip Time
SCION	Scalability, Control, and Isolation on Next-Generation Networks
SDN	Software-Defined Networking
SR	Segment Routing
SRv6	Segment Routing over IPv6
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TNA	Tofino Native Architecture
X.509	Public-key certificate standard used by TLS and mTLS

Symbols

Symbol	Definition	Unit
G	Public generator for optical DH key exchange	—
P	Public random number for optical DH key exchange	—
A, B	Private secrets of Hermes and switch respectively	—
K_A, K_B	Public keys: $(G \oplus P)$ & A, $(G \oplus P)$ & B	—
S	Shared secret: $(K_B$ & $A) \oplus P$	—
acc	Packet accumulator field	32-bit

Symbol	Definition	Unit
flow_id	Identifier for a Hermes flow or path session	32-bit
I	Current key index	—
L	Path length (number of hops)	hops
N_k	Total number of keys per switch per DH exchange	—
op_I	Opcode assigned to the key at index I	3-bit
seq	Per-flow monotonically increasing sequence number	32-bit
σ	LFSR seed distributed during key provisioning	32-bit
timestamp_ms	Millisecond-resolution packet timestamp used for replay protection	48-bit
ν	Per-flow nonce issued by the Hermes server and used as accumulator seed	32-bit
t_d	Digest receipt latency	ms
t_{DH}	DH key-exchange round-trip time	ms
t_v	End-to-end verification latency	ms

1 Introduction

The modern Internet gives end users little visibility into, or control over, the paths their packets take. When an organisation or individual sends data from one host to another, the route is determined by network operators and routing protocols, most notably BGP at the inter-domain level. The sender can choose the destination, but not which networks or routers the packet should traverse on the way there. Even if a sender could express such a preference, there is still a second problem: the sender cannot later verify that the packet actually followed the intended path. This lack of path control and path visibility is increasingly problematic. Geopolitical boundaries, regulatory requirements, corporate security policies, and attacks on routing protocols, such as BGP hijacking, all give users strong reasons to care not only about whether data arrives, but also about *where it travelled* before it arrived [26, 19].

Consider an organisation that is required to keep sensitive traffic within a specific jurisdiction, or that wants traffic to pass only through approved providers and inspection points. A routing system may allow the organisation to request such a path, for example by using source routing, Segment Routing, or an SDN controller. However, path selection alone is not sufficient. If the network cannot later prove that the packet actually followed the selected path, then the control mechanism depends entirely on trust. A malicious, compromised, or misconfigured router could silently skip an inspection point, insert an unauthorised hop, or redirect traffic through an untrusted network. The sender would still see that the packet arrived, but would have no evidence that the route was honoured. This makes path verification a necessary complement to controllable routing: without verification, user-defined routing policies cannot be audited or enforced after the fact.

This thesis investigates whether packet forwarding can be made both controllable and verifiable without relying on heavyweight cryptography in the data plane. The central research question is:

> *Can we achieve an efficient and secure routing protocol that enables user-defined, controllable packet forwarding with cryptographic post-transmission path verification, and with minimal overhead?*

Two practical challenges follow from this question. First, the verification evidence must not grow significantly with the number of hops. A design that appends a separate token, signature, or telemetry record at every switch quickly becomes expensive on longer paths. Second, verification must be compatible with high-speed forwarding. Standard cryptographic primitives such as AES, HMAC, and classical Diffie–Hellman provide strong security guarantees, but are poorly suited to a P4 match-action pipeline. Programmable switches are used in this thesis precisely because they make it possible to perform custom per-packet processing at line rate, but only if the per-hop operation is simple enough to fit within the forwarding pipeline.

To answer the research question, this thesis proposes **Hermes**, a path-verification system for programmable data planes. Hermes lets a sender use a selected sequence of trusted switches and later obtain cryptographic evidence that a packet traversed that sequence. Rather than attaching a growing list of per-hop proofs to the packet, Hermes uses a compact accumulator field that is updated at each authorised switch. The final accumulated value is reported to a trusted Hermes server, which

independently reconstructs the expected computation and returns an Accept or Reject verdict. In this way, Hermes provides post-transmission path verification while keeping the verification evidence constant in size.

At a high level, Hermes separates the data-plane and control-plane responsibilities. The P4 data plane performs only lightweight accumulator updates: each switch looks up its current secret key and opcode, applies the corresponding bitwise or fixed-width arithmetic operation to the accumulator, increments the hop counter, and forwards the packet. The control plane is responsible for heavier tasks: provisioning per-switch key pools, opcode assignments, and LFSR seeds; protecting the provisioning channel with mutual TLS; receiving the final digest; and asking the Hermes server to verify the reported accumulator. This separation keeps the forwarding path simple while still allowing the server to check, after transmission, whether the packet followed the authorised sequence of switches.

Hermes is designed around fixed overhead and bounded data-plane work. Each packet carries a fixed Hermes header containing the accumulator, flow identifier, hop count, sequence number, timestamp, and nonce. The header does not grow with path length: every switch folds its contribution into the same accumulator field. Replay protection is provided by per-flow sequence numbers and a timestamp window, while the nonce binds packets to the server-issued flow session and seeds the initial accumulator. Key reuse is managed by a key lifecycle in which switches first consume a pool of keys sequentially and then use an LFSR-driven reuse order until renegotiation is triggered.

The prototype evaluation shows that Hermes is feasible within the tested programmable data-plane environment. Key provisioning remains stable across the tested key-pool sizes, end-to-end verification latency remains approximately constant across the tested packet rates. The forwarding experiments show that the measured throughput is limited by the software packet generator rather than by the switch pipeline, and the flow-completion experiment records the offered workload without packet loss. Hermes also correctly rejects all tested path-deviation and replay scenarios, including missing-hop, extra-hop, stale-timestamp, replayed-packet, and corrupted-accumulator cases.

The security analysis shows that Hermes provides meaningful path-integrity evidence under the stated threat model. An adversary that skips, inserts, reorders, or modifies hops will generally produce an accumulator value that does not match the server's independent replay. A single unknown hop contribution is protected by opcode ambiguity, while sequence numbers and timestamps prevent replay of previously accepted packets. The main security trade-off is that Hermes deliberately chooses a lightweight P4-compatible accumulator rather than a conventional MAC chain. This makes the system deployable in the data plane, but means that key reuse must be bounded and renegotiation must occur before repeated observations reveal too much information. The control-plane provisioning channel is therefore protected with mutual TLS, and the remaining limitations are identified explicitly in the security analysis and discussion.

This thesis is organised as follows. Chapter 2 introduces the networking, programmable-data-plane, and cryptographic background needed to understand Hermes. Chapter 3 surveys related work in routing control, packet-level path verification, and programmable network security. Chapter 4 presents the Hermes design. Chapter 5 analyses the security of the system. Chapter 6 describes the evaluation methodology and experimental results. Chapter 7 discusses the scope, limitations, and directions for future work, and Chapter 8 concludes the thesis.

2

Background

This chapter provides the technical foundation needed to understand the design and security properties of Hermes. It covers three areas, each directly mapped to a part of the system.

Networking and routing (Section 2.1) explains the Internet's routing infrastructure and its trust problem: packets can silently deviate from their intended path, and today's protocols provide no mechanism to detect this. That gap is the primary motivation for Hermes.

Programmable data planes and P4 (Section 2.2) describes the P4 language and the programmable switch technology on which Hermes runs. Understanding P4's capabilities and constraints is essential for understanding why the accumulator is designed the way it is.

Cryptographic foundations (Section 2.3) introduces the primitives that Hermes uses (optical Diffie–Hellman key exchange, linear feedback shift registers, and replay-protection techniques) and explains why stronger but heavier alternatives, e.g. AES, HMAC, and classical Diffie–Hellman, cannot be used in the data plane.

Readers familiar with a particular area may skip the corresponding section, as each is written to be self-contained.

2.1. Networking and Routing

Hermes operates inside an existing network infrastructure. This section explains how Internet routing works and why it creates a path-integrity problem that Hermes addresses. It introduces Autonomous Systems and the trust relationships between them, the three main families of routing protocols (both intra-domain and inter-domain), the source-routing paradigm that Hermes builds on, and the specific path-visibility gap that motivates the system.

2.1.1. Autonomous Systems and the Internet's Trust Model

The Internet is not a single monolithic network but a collection of independently administered networks called *Autonomous Systems* (ASes). Each AS is a collection of IP address prefixes and routers managed by a single administrative entity, e.g., an Internet Service Provider (ISP), a university, a CDN, or a large enterprise, and identified by a globally unique *Autonomous System Number* (ASN) assigned by regional Internet registries [27].

ASes interact with one another under bilateral or multilateral agreements governed by one of three relationship types:

- **Provider–customer:** the customer pays the provider for transit, i.e., reachability to the rest of the Internet.
- **Peer–peer:** two ASes agree to exchange traffic between their respective customers at no charge. Neither AS will carry the other's traffic to a third party.

- **Sibling–sibling:** two ASes under the same administrative umbrella exchange all routes freely.

These relationships shape which routes each AS is willing to announce to which neighbours, a set of rules known as the *valley-free* routing model [7]. The result is that the path a packet takes between two hosts on the Internet is determined by the routing decisions of every AS along the way, not by the sender or receiver.

Routing can be divided into two scopes: *intra-domain* routing, which determines the path *within* a single AS, and *inter-domain* routing, which determines the path *between* ASes. The following subsection covers both.

This architecture creates a fundamental trust problem. There is no mechanism that forces AS i to forward a packet through the path that AS $i-1$ intended. The attack where an AS falsely announces ownership of an IP prefix it does not control is known as *BGP hijacking*, and it has been responsible for high-profile incidents, including the 2010 China Telecom event that redirected approximately 15% of the world's Internet routes through China for 18 minutes [19]. The absence of path enforcement at the IP layer is the root motivation for the research this thesis addresses.

2.1.2. Routing Protocols

Routing protocols determine the next hop for each packet. We cover three families: intra-domain distance-vector (RIP), intra-domain link-state (OSPF), and inter-domain path-vector (BGP). A consistent theme across all three is that routing decisions are made independently at each router based on locally computed metrics or received advertisements, with no cryptographic guarantee that packets follow the advertised path.

Distance-vector routing: RIP. The *Routing Information Protocol* (RIP) is one of the oldest intra-domain routing protocols [11, 1]. Each router periodically broadcasts its entire routing table to directly connected neighbours. Routes are selected based on hop count, and routes exceeding 15 hops are declared unreachable.

The simplicity of RIP creates well-known problems. When a link fails, routers continue advertising stale routes, causing *counting-to-infinity* loops that take many update cycles to resolve. RIP's single metric also ignores link capacity, delay, and reliability. For these reasons, RIP is limited to small, stable networks.

Link-state routing: OSPF. *Open Shortest Path First* (OSPF) is the dominant intra-domain link-state protocol [21]. Rather than exchanging routing tables, each OSPF router floods *Link-State Advertisements* (LSAs) describing its links and costs. All routers build a complete, synchronised copy of the *Link-State Database* (LSDB) and run Dijkstra's algorithm independently to compute shortest paths.

OSPF supports *areas* to limit the scope of LSA flooding, and equal-cost multi-path routing (ECMP) to load-balance across parallel paths. OSPF converges within seconds after a link failure and supports richer metrics than RIP. However, like all intra-domain protocols, OSPF has no concept of path trust: the shortest path is always chosen regardless of which routers it traverses.

Path-vector routing: BGP. The *Border Gateway Protocol* (BGP) [26] is the sole inter-domain routing protocol of the Internet. Unlike RIP and OSPF, BGP makes routing decisions based on *policies* rather than a single metric. Each route advertisement includes the `AS_PATH` (the sequence of ASes traversed), and policy attributes such as `LOCAL_PREF` (local preference among routes to the same destination), `MED` (influencing inbound traffic), and `COMMUNITY` (tags for policy signalling). A BGP speaker selects the route that best satisfies its locally configured policies, not necessarily the shortest or fastest path.

BGP's critical security weakness is that it has no built-in mechanism to verify that an AS advertising a prefix actually owns it, or that the `AS_PATH` reflects the true forwarding path. Resource Public Key Infrastructure (RPKI) addresses origin validation and prevents BGP hijacking at the prefix-ownership level, but does not prevent path manipulation [19]. Even with RPKI, the data plane (actual packet forwarding) is entirely separate from the control plane (BGP advertisements): an AS can advertise a legitimate path while silently forwarding packets via a different route. This data-plane/control-plane gap is precisely the problem that Hermes addresses at the switch level.

2.1.3. Source Routing and Segment Routing

Source routing is the forwarding paradigm closest to what Hermes requires: the sender, or a trusted controller acting on its behalf, specifies the forwarding path in the packet header rather than delegating the decision to intermediate routers. Hermes builds on this paradigm but adds a cryptographic layer that existing source-routing schemes lack.

Classical source routing. IPv4 provides two source-routing options in the IP options header. *Strict source routing* (SSR) lists every router the packet must traverse in order. *Loose source routing* (LSR) lists waypoints with unconstrained hops between them. Both options are widely blocked by ISPs because SSR can be used to bypass firewalls, and both add header overhead.

Segment Routing. *Segment Routing* (SR) [6] is a modern, scalable realisation of source routing widely deployed in ISP backbones and data-centre fabrics. SR encodes an ordered list of *segments* in the packet header; each segment is a forwarding instruction such as "forward to node X" or "apply service function Z".

SR comes in two data-plane flavours. **SR-MPLS** encodes segment identifiers as MPLS labels. MPLS (Multi-Protocol Label Switching) is a traffic-engineering mechanism that uses short labels placed on top of layer 2 to forward packets along pre-determined paths; routers swap or pop the top label at each hop. **SRv6** encodes segments as 128-bit IPv6 addresses in a *Segment Routing Header* (SRH); each hop decrements a pointer and updates the destination address to the next segment.

SR establishes that switch-level explicit path control is operationally deployed at scale. However, SR provides no cryptographic assurance that the packet actually followed the segment list; an on-path adversary can silently modify the SRH or reroute packets without detection. Hermes's accumulator scheme (Section 4.4) provides exactly this missing cryptographic assurance.

2.1.4. The Path-Visibility Problem

The combination of BGP's control-plane/data-plane gap and the absence of cryptographic path enforcement means that neither senders nor receivers have reliable visibility into which physical routers their packets traversed. This has practical consequences:

- Organisations subject to data-residency regulations cannot verify their traffic stayed within a particular jurisdiction.
- Security-conscious senders cannot confirm their traffic avoided known-malicious or state-controlled ASes.
- Post-incident forensics cannot determine whether a deviation was caused by misconfiguration, a BGP hijack, or deliberate interception.

The goal of Hermes is to provide a cryptographic proof of path traversal at the intra-domain programmable switch level, as a deployable stepping stone toward broader Internet path integrity.

2.2. Programmable Data Planes and P4

P4 is a domain-specific language for describing how network switches process packets. Unlike traditional fixed-function switches, a P4 switch can be programmed to parse and forward any protocol the operator defines, without hardware modifications. Hermes deploys a P4 program on all switches to perform per-hop accumulator updates. Two properties of P4 are critical for the design: its *capabilities* (custom headers, match-action tables, stateful registers, and digest emission) make it possible to maintain per-flow key state and report a final accumulator value to the controller; its *constraints* (no loops, no modular arithmetic, no native cryptographic primitives) rule out standard approaches such as AES-based MACs and classical Diffie–Hellman, and directly motivate Hermes's lightweight design choices.

P4: key properties. P4 [2] is designed around four principles: *reconfigurability* (the forwarding behaviour can be changed after deployment by loading a new compiled P4 program); *protocol independence* (all packet formats are user-defined, with no built-in assumptions); *target independence* (the same program

compiles to different hardware or software targets); and *deterministic execution* (no loops or dynamic memory allocation, making the pipeline statically analysable).

At its core, a P4 programmer defines the bit-level layout of every packet header, a parser that extracts those headers from the byte stream, a *match-action pipeline* where tables match on header fields and execute actions, and a deparser that reassembles the modified packet for transmission. Tables are populated at runtime by a controller via P4Runtime; the program defines only the schema. In Hermes, the `share_table` and `opcode_table` hold per-key secrets installed by the controller after each Diffie–Hellman exchange.

Registers and digests. Two additional primitives are central to Hermes. *Registers* are stateful arrays that persist between packets, allowing per-flow state such as key indices and packet counters to be maintained inside the switch. *Digests* let the data plane notify the controller about a packet event by emitting a small struct asynchronously. Hermes uses registers to track key state at each switch, and a digest to report the final accumulator value to the controller at the last hop.

Constraints. P4's determinism requirement means there are no loops, no dynamic memory allocation, and no native modular arithmetic or cryptographic primitives. As a result, implementing cryptographic operations in P4 requires algorithms that rely solely on simple bitwise logic and fixed-width integer arithmetic, avoiding expensive operations like modular exponentiation or hash function evaluation. This constraint shapes the entire cryptographic design of Hermes.

Hardware and software targets. P4 programs can be compiled to both software and hardware targets. The *BMv2* (Behavioural Model version 2) software switch simulates the P4 pipeline in C++ and is widely used for prototyping and debugging P4 programs. Software targets are useful during development because they are easy to instrument and run on commodity machines, but their packet-processing rate is limited by the host CPU and therefore does not represent production forwarding performance.

Hardware targets execute P4 programs in a specialised packet-processing pipeline. In this thesis, Hermes is evaluated on an Intel Tofino ASIC using the Tofino Native Architecture (TNA). This is relevant because Tofino executes the data-plane operations in hardware, so the measured forwarding behaviour reflects a realistic programmable switch target rather than a software simulator. The Hermes data-plane design is therefore constrained by what can be expressed in a hardware P4 pipeline: fixed-width header parsing, match-action tables, register accesses, digest emission, and simple integer or bitwise operations. These constraints directly motivate the use of a compact accumulator and lightweight per-hop operations instead of AES, HMAC, or modular exponentiation in the forwarding path.

P4Runtime. P4Runtime [2] is the vendor-neutral, gRPC-based control interface for P4 switches. It allows a controller to push a new P4 program atomically, install and update match-action table entries, read and write register values, and subscribe to digest notifications. Hermes's controller (`run_hermes.py`) uses all four operations: it installs the forwarding rules and per-key assignments after each DH exchange, writes the LFSR seed to registers, and receives digest notifications carrying the final accumulator value.

In-band Network Telemetry. *In-band Network Telemetry* (INT) [15] is the direct architectural predecessor of Hermes's accumulator design. INT embeds per-hop metadata directly into packets: each transit switch appends a record (containing switch ID, queue depth, timestamps, etc.) to a growing header stack, and a sink switch strips the full stack and forwards it to a monitoring backend. The total header overhead is $L \times |\text{record}|$ bytes for a path of length L.

Hermes is *inspired by* this per-hop appending pattern but makes two critical modifications. First, instead of concatenating records (growing the header $O(L)$), Hermes folds each hop's contribution into a single 32-bit field using a mathematical operation, keeping overhead $O(1)$. Second, each hop's contribution is *keyed*: it depends on a secret known only to that switch and the Hermes server, so an on-path attacker cannot forge it, whereas any INT record can be freely modified.

2.3. Cryptographic Foundations

This section introduces the cryptographic primitives relevant to Hermes. For each one it explains what it provides, whether it is suitable for a P4 data plane, and what role it plays in the system. Some mechanisms, such as TLS and mutual TLS, are not used inside the P4 pipeline at all; they secure the software control plane that provisions the data-plane keys.

2.3.1. Symmetric Cryptography

In symmetric-key cryptography, a single secret key k is shared between parties and used for both encryption and authentication:

$$c = E_k(m), \qquad m = D_k(c).$$

The security guarantee is that an adversary observing (c, E_k, D_k) cannot recover m without k in polynomial time.

The most common symmetric schemes, AES block encryption and HMAC message authentication,provide strong security guarantees and are the standard per-hop primitives in path-verification systems such as SCION, ICING, and L-SPV (Section 3). However, both require operations (substitution-permutation rounds and hash function evaluations, respectively) that are not natively expressible in a P4 match-action pipeline; implementing them in P4 requires large pre-computed lookup tables and is prohibitively expensive per packet. *Pseudo-random functions* (PRFs), similarly built from block ciphers or hash functions, share the same deployability constraint. For this reason, Hermes replaces these standard primitives with a simpler XOR/AND-based accumulator, accepting a weaker per-observation security bound in exchange for a data-plane implementation that requires no cryptographic extern functions.

2.3.2. Asymmetric Cryptography

Public-key (asymmetric) cryptography uses a key pair (pk, sk): pk is public, sk is secret. Encryption uses the public key; decryption requires the private key. The security of asymmetric schemes rests on computational hardness assumptions such as the difficulty of factoring large integers (RSA) or the elliptic-curve discrete logarithm. Asymmetric operations involve modular exponentiation over large integers (2048-bit or more), which is several orders of magnitude slower than AES and entirely incompatible with P4's data-plane execution model.

2.3.3. Classical Diffie–Hellman Key Exchange

The *Diffie–Hellman* (DH) key exchange [5], introduced in 1976, allows two parties to agree on a shared secret over an insecure channel without any prior shared secret. Public parameters are a large prime p and a generator g. Alice generates a secret a, sends $u = g^a \bmod p$; Bob generates b, sends $v = g^b \bmod p$; both compute the shared secret $s = g^{ab} \bmod p$. An eavesdropper observing g, p, u, v cannot recover s without solving the discrete logarithm problem, which has no known polynomial-time solution for appropriately chosen p.

Classical DH is introduced here primarily to motivate the optical variant that Hermes actually uses: it provides the strongest key-agreement security guarantees, but its core operation, modular exponentiation over 2048-bit integers, is not expressible in P4's match-action primitives, requiring a lightweight substitute.

2.3.4. Optical Diffie–Hellman (XOR-based Variant)

Jeon and Gil [13] propose replacing modular exponentiation with XOR and AND operations. Both operations are directly expressible in P4, making this variant suitable for the data plane.

Protocol. Public parameters are two random 32-bit values G and P. Alice (the Hermes server) chooses a random secret A and sends $K_A = (G \oplus P) \,\&\, A$; Bob (the switch) chooses B and sends $K_B = (G \oplus P) \,\&\, B$. Both compute the shared secret:

$$S = (K_B \,\&\, A) \oplus P = (K_A \,\&\, B) \oplus P = \big((G \oplus P) \,\&\, A \,\&\, B\big) \oplus P.$$

Equality holds because AND is commutative. An eavesdropper observing G, P, K_A, K_B cannot recover S without knowing A or B: the AND operation destroys all information about the secret on the bits

where $G \oplus P = 0$, making recovery impossible on those positions. The security guarantee holds as long as A and B are random and never reused. Importantly, because XOR and AND are the only operations required, this key exchange can be implemented entirely in P4, enabling Diffie–Hellman key agreement within the data plane.

In Hermes, this protocol is used to establish a *list* of shared keys between the controller and the Hermes server, along with associated operations and an LFSR seed. The details of how these are used are explained in Section 4.3.

2.3.5. Linear Feedback Shift Registers

A *Linear Feedback Shift Register* (LFSR) is a simple pseudo-random sequence generator based on a shift register whose feedback is a linear (XOR) combination of selected bits. LFSRs are extremely cheap to implement, an n-bit LFSR requires only n flip-flops and a few XOR gates, making them suitable for use inside a P4 switch.

An n-bit LFSR has a state $s \in \{0, 1\}^n$. At each step, it shifts right by one bit, and the new high bit is:

$$s_{n-1} = \bigoplus_{j \in T} s_j,$$

where T is the *tap set* (the feedback polynomial). If the feedback polynomial is a *primitive polynomial* over GF(2), the LFSR cycles through all $2^n - 1$ non-zero states before repeating — a *maximal-length* (m-)sequence with near-uniform statistics.

In Hermes, a 32-bit Galois LFSR is used to drive pseudo-random key-index selection after the initial key pool has been consumed sequentially. The exact polynomial and update rule are given in Section 4.5. The LFSR is *not* used as a cryptographic primitive: it is used only as a lightweight index-permutation mechanism to make the key-reuse sequence unpredictable to an adversary who does not know the seed.

2.3.6. Message Authentication Codes

A *Message Authentication Code* (MAC) is a short tag computed from a message m and a secret key k:

$$t = \mathrm{MAC}_k(m).$$

A recipient who knows k can verify the tag; an adversary without k cannot forge a valid (m, t) pair (*unforgeability under chosen-message attack*). MACs differ from hash functions in that a hash $H(m)$ requires no key and provides no authentication guarantee. HMAC, the standard MAC construction, is built from a cryptographic hash function and is used in TLS 1.3, IPsec, and SSH.

MACs are the standard per-hop primitive in existing path-verification systems such as SCION, ICING, and L-SPV, providing strong unforgeability guarantees. As discussed above, however, HMAC and other MAC constructions require hash function evaluations that are not expressible in a P4 pipeline without expensive pre-computed tables. This is the fundamental reason Hermes cannot use standard MACs for per-hop verification, and why it accepts the weaker $1/8$ opcode-guessing bound of the XOR/AND accumulator in exchange for data-plane deployability.

2.3.7. Replay Attacks and Defences

A *replay attack* occurs when an adversary captures a legitimately transmitted message and retransmits it later to cause the recipient to take an action that should only happen once. Three standard countermeasures are used in combination to prevent this.

Sequence numbers are monotonically increasing per-flow counters. The verifier rejects any message with a sequence number no greater than the highest previously accepted, preventing replay of any past message.

Timestamps embed the sender's current time in the message. The verifier rejects messages whose timestamp falls outside a window δ of its own clock, limiting the replay window to 2δ even if per-flow state is lost. This requires clock synchronisation between sender and verifier.

Nonces are random values generated freshly for each message. The verifier maintains a set of seen nonces within the timestamp window and rejects duplicates. Unlike sequence numbers, nonces do not require the verifier to maintain a counter, but they do require bounded storage for the nonce set.

Used together, these three techniques block replays even when any individual mechanism fails: a replayed message with the same sequence number is rejected by the counter check; a message with a modified sequence number requires knowledge of the secret to produce a matching authenticator; and a stale timestamp is rejected even if the sequence check has been reset. How these three mechanisms are instantiated in Hermes is described in Section 4.6.

2.3.8. Transport Layer Security and Mutual TLS

Transport Layer Security (TLS) is the standard protocol for securing communication between two software endpoints over an untrusted network [28]. It provides three main properties. First, it provides *confidentiality*: after the handshake, application data is encrypted and cannot be read by a passive observer. Second, it provides *integrity*: an attacker cannot modify protected messages without detection. Third, it provides *endpoint authentication*: at least one side, usually the server, proves its identity using a certificate.

TLS operates above TCP and below the application protocol. This means that the application can keep its own message format while the transport channel underneath it is encrypted and authenticated. In Hermes, this distinction is important. The P4 data plane does not run TLS and does not perform public-key cryptography. Instead, TLS is used only for the control-plane communication between the Hermes server and the switch control-plane components. In ordinary server-authenticated TLS, the server presents a certificate and the client verifies it. This prevents a controller or switch control-plane component from sending sensitive provisioning data to an impostor server. *Mutual TLS* (mTLS) extends this model by requiring the client to present a certificate as well. Both endpoints therefore authenticate each other: the client verifies that it is connected to the real Hermes server, and the server verifies that the request came from an authorised Hermes control-plane component. The certificates used for this purpose follow the X.509 public key infrastructure model [4].

For Hermes, mTLS is used as a control-plane hardening mechanism. It protects the key pool, opcode assignments, and LFSR seed while they are transported between the server and the switch control plane, and it prevents unauthorised hosts from initiating provisioning, path-registration, or verification requests. This does not change the per-packet data-plane design: switches still execute only the lightweight accumulator operations described in Chapter 4. The cost of mTLS is therefore limited to the software control plane and does not add packet-header overhead, P4 pipeline stages, or per-hop data-plane cryptographic operations.

2.4. Threat Model

We assume an on-path (man-in-the-middle) adversary located anywhere between h_1 and h_2, including on inter-switch links. The adversary's goal is to make a packet that did *not* traverse the authorised path pass verification, or to learn a switch's secret key so that it can forge contributions at will. The attacker may:

- intercept, drop, delay, reorder, and replay packets;
- inject new packets;
- read or modify any unencrypted packet field;
- observe the accumulator entering and leaving a switch (by tapping the links immediately before and after it).

The attacker *cannot*:

- access the Hermes server or any switch's internal state or memory;
- obtain secret keys or LFSR seeds;
- break XOR/AND-based operations given random, non-reused keys.

Switches are trusted to execute their P4 program correctly but are not under our administrative control. The control / verification plane is assumed authenticated. In the current prototype, the controller-to-server channel is protected with mutual TLS, so Hermes provisioning and verification messages are encrypted and both endpoints authenticate each other. The controller-to-switch P4Runtime channel and sender-to-server path request channel are treated as trusted control-plane channels and are outside the data-plane attacker model. Our goal is to ensure *integrity of path-verification evidence*: an adversary cannot make an injected or modified packet pass verification without the required cryptographic material. Availability (e.g. packet loss and delay) is out of scope.

3

Related Work

Research relevant to this thesis spans three broad areas: architectural frameworks for routing control and trust, packet-level path verification protocols, and security mechanisms for programmable data planes. This chapter surveys the most pertinent contributions in each area, going into technical detail where the mechanisms are directly comparable to or influential on the design of Hermes, and concludes with a positioning table.

3.1. Routing Control and Trust Architectures

3.1.1. Responsible Internet

Hesselman et al. articulate a vision for a *Responsible Internet* [12] in which network operators and end users are empowered with visibility, accountability, and policy control over data flows. Their framework argues that today's Internet lacks adequate mechanisms for users to express preferences about which networks their traffic traverses, and that operators lack the tooling to enforce or audit those preferences. The proposed architecture introduces *routing transparency services* that publish path decisions and *policy enforcement points* at network boundaries that can reject or redirect traffic violating stated preferences.

Technically, the framework does not define a specific wire protocol; instead it surveys the system properties required of any responsible routing infrastructure: *verifiability* (can the user confirm the path was followed?), *accountability* (can deviations be attributed?), and *enforceability* (can the network actually honour the stated policy?). These three properties map directly onto the design requirements of Hermes: the accumulator scheme provides verifiability, the cryptographic binding of each hop's contribution provides accountability, and the P4 data-plane implementation provides enforceability at the switch level. While the Responsible Internet vision remains largely architectural, it provides a useful architectural motivation for user-controlled routing research.

3.1.2. User Controlled Internet Protocol (UCIP)

Kheirkhah et al. propose the *User Controlled Internet Protocol* (UCIP) [14], which allows end users to attach routing directives directly to packets expressing preferences about which ASes the packet should or should not traverse.

Mechanism. UCIP extends the IP header with a *policy field* encoding an ordered list of AS-level constraints. Border routers interpret these constraints and select routes that satisfy them from their local routing information base; if no conforming route exists, the packet is forwarded along the best available path and a *violation flag* is set. Cryptographic enforcement is provided through *policy tokens*: each AS on the intended path pre-commits to a cryptographic token that the sender embeds in the packet. An AS can only validate its own token, not its neighbours', preventing downstream ASes from forging evidence of upstream traversal. At the destination, the token chain is checked against the declared policy, and any missing or invalid token signals a deviation.

Comparison to Hermes. UCIP demonstrates that decentralised per-packet policy expression is practically feasible and that verification does not require a centralised authority. Hermes addresses a simpler but more immediately deployable scenario: a single administrative domain with programmable switches, where full enforcement is possible at each hop without ISP co-operation, trading UCIP's AS-scale deployment scope for tighter, switch-granularity guarantees. Where UCIP distributes tokens to ASes before a session begins, Hermes distributes keys to switches during the DH exchange and does not require any per-flow setup with external parties.

3.1.3. Mutually Controlled Routing (MCR)

Mahajan et al. introduce *Mutually Controlled Routing* (MCR) [20], which distributes path-enforcement responsibility among multiple independent ISPs. The central insight of MCR is that end-to-end path enforcement in the Internet cannot be achieved by a single party: it requires each AS on the path to independently verify and enforce its own segment.

Mechanism. MCR decomposes a source-specified path into *route fragments*, each owned by one ISP. The source cryptographically commits to the full intended path using a chain of tokens, one per AS. Each AS validates the token for its segment before forwarding and refuses to forward packets whose tokens are invalid or do not match the locally configured route fragment. The token chain is constructed so that no single AS can forge or reuse a token intended for a different position in the path, even if it has observed previous traffic. Cryptographic binding between adjacent tokens in the chain ensures that the overall path cannot be re-assembled from tokens collected from multiple separate sessions.

Comparison to Hermes. MCR's per-hop token assignment is conceptually similar to Hermes's per-hop key assignment: each switch on the path holds a secret that it contributes to the verification evidence, and no switch can substitute another switch's contribution. MCR operates at AS granularity with public-key token chains, while Hermes operates at switch granularity with symmetric-key accumulator operations designed for the P4 data plane. The route-fragment decomposition also foreshadows the design principle that path proof should be composable from independent per-hop contributions.

3.1.4. Pathlet Routing

Godfrey et al. introduce *Pathlet Routing* [8], a flexible interdomain routing scheme in which the routing space is decomposed into reusable *pathlets* — short, composable path fragments with defined entry and exit points. Sources construct end-to-end routes by chaining pathlets drawn from a global pathlet graph published by ISPs.

Mechanism. Each ISP advertises a set of pathlets it is willing to provide, each identified by a compact identifier. The source embeds an ordered list of pathlet identifiers in the packet header. Routers match the current pathlet identifier against a local forwarding table and forward accordingly, advancing to the next identifier at each AS boundary. The composition model allows fine-grained path expression, including avoidance of specific ASes and preference for low-latency links, without requiring the source to have full topology visibility.

Comparison to Hermes. While Pathlet Routing is not cryptographic in nature and does not provide post-transmission verification, its decomposition principle directly informs Hermes's per-hop key assignment: each forwarding element contributes independently to the overall path proof, mirroring the way each pathlet contributes independently to the overall forwarding path. The scalability analysis in Pathlet Routing also supports the observation that per-hop state need not be large to enable expressive path control.

3.1.5. Outsourcing Routing Control via SDN

Kotronis et al. propose offloading interdomain routing control logic to SDN-style controllers [17], separating the forwarding decision from the route-computation logic. The controller computes routes subject to user-defined policies and installs forwarding rules via a southbound API, enabling dynamic, flexible enforcement without per-hop deep packet inspection.

This architectural pattern aligns closely with Hermes's design: the Hermes server retains all cryptographic state and verification logic while the P4 switches perform only lightweight per-packet accumulator updates. The controller in the Kotronis model plays a role analogous to the Hermes server — it holds the policy view and enforces it by programming the data plane, rather than by processing every packet itself. Separating verification intelligence from forwarding hardware is a key scalability principle that both systems share.

3.1.6. SCION

The most comprehensive architecture in this space is SCION (Scalability, Control, and Isolation on Next-Generation Networks) [25], a clean-slate interdomain routing architecture designed to address the security, availability, and path-control limitations of today's Internet.

Trust model. SCION organises the Internet into *Isolation Domains* (ISDs), each governed by a set of *core ASes* responsible for *Trust Root Configuration* (TRC) files that define the cryptographic trust anchors and policies within the domain. This federated model prevents any single certificate authority from compromising global routing and limits the blast radius of a key compromise to a single ISD.

Path construction. Routing is based on *Packet-Carried Forwarding State* (PCFS), where paths are precomputed and embedded in packet headers. A path consists of three concatenated segments: an *up-segment* from the source to an ISD core, a *core-segment* between ISD cores, and a *down-segment* from the ISD core to the destination. Each segment is a sequence of *Hop Fields* (HFs), each containing the ingress and egress interface identifiers for one AS and a per-hop MAC:

$$\mathrm{MAC}_i = \mathrm{PRF}_{K_i}\big(t_{\mathrm{exp}} \,\|\, \mathrm{IF}_{\mathrm{in}} \,\|\, \mathrm{IF}_{\mathrm{out}} \,\|\, \mathrm{MAC}_{i-1}\big),$$

where K_i is a symmetric key shared between the path server and AS i, t_{exp} is a path expiry timestamp, and $\mathrm{IF}_{\mathrm{in/out}}$ are the interface identifiers. This chained MAC construction means each AS can verify that the immediately preceding hop was legitimate without needing to know the keys of other ASes. An AS that is not on the intended path cannot produce a valid HF, and any packet with an invalid HF is silently dropped.

Comparison to Hermes. SCION's per-hop MAC chain is structurally the closest prior mechanism to Hermes's per-hop accumulator: both build a cryptographic proof of traversal by having each hop apply a secret keyed operation to a value carried in the packet header. The key differences are scope and implementation. SCION targets the full Internet and requires a clean-slate trust infrastructure; Hermes targets intra-domain P4 switches and operates over existing hardware without modifying the surrounding network. SCION's MAC uses a standard PRF, which requires modular arithmetic not available natively in P4; Hermes replaces this with the XOR/AND-based optical DH accumulator that is directly expressible in a P4 match-action pipeline without cryptographic extern functions. SCION also does not address the key-reuse and opcode obfuscation mechanisms that Hermes introduces to resist traffic analysis on the accumulator values.

3.2. Packet-Level Path Verification

3.2.1. ICING

Naous et al. present the *ICING* protocol [22], which provides both path enforcement and post-transmission verification for interdomain routing. ICING allows a sender to explicitly specify the AS-level path a packet should take and to attach cryptographic tokens that each AS must validate before forwarding.

Token construction. ICING uses a *nested MAC* structure. Before sending, the source obtains *capability tokens* from each AS on the intended path. The token T_i for AS i is computed as:

$$T_i = \mathrm{MAC}_{K_i}(\mathrm{src} \,\|\, \mathrm{dst} \,\|\, \mathrm{path_id} \,\|\, T_{i-1}),$$

where K_i is a key shared between the source and AS i, and T_{i-1} is the token of the immediately preceding AS (with T_0 being a source-generated nonce). The chain construction ensures that AS i can verify only

its own token and the preceding one, keeping per-AS processing cost constant regardless of path length. Because each token commits to T_{i-1}, tokens cannot be reordered or combined from different paths: doing so would break the chain and cause verification to fail.

Path enforcement and verification. Each AS along the path validates its token before forwarding, and drops packets with invalid tokens. The destination receives the full token chain and forwards it to a verifier (analogous to the Hermes server), which checks that each token is valid and that the chain corresponds to the intended path. Any deviation — a missing token, a forged token, or a token from an unexpected AS — causes verification to fail.

Comparison to Hermes. ICING's per-hop contribution paradigm is foundational for Hermes: each hop contributes unforgeable evidence of traversal, and the evidence is verified centrally after transmission. Hermes adopts the same structural pattern but replaces the MAC chain with a single 32-bit accumulator field updated by a keyed bitwise operation. This substitution has two practical consequences: (i) the per-packet header overhead drops from $O(L)$ tokens (one per hop) to $O(1)$ (one 32-bit field), and (ii) the per-hop operation is expressible in a P4 match-action pipeline without any cryptographic extern functions, making the scheme deployable on the evaluated Tofino target using P4-compatible operations.

3.2.2. Lightweight Source Authentication and Path Validation (L-SPV)

Kim et al. introduce *L-SPV* [16], providing source authentication and end-to-end path validation at Internet scale while minimising computational and bandwidth overhead. The design avoids public-key cryptography entirely, relying on symmetric-key PRFs and an efficient key-derivation strategy.

Path Validation Field. The central data structure is a *Path Validation Field* (PVF), a fixed-size header field that accumulates authentication information as the packet traverses the path. Each AS i updates the PVF using a PRF with its pairwise key $K_{s,i}$ shared with the source:

$$\mathrm{PVF}_i = \mathrm{PRF}_{K_{s,i}}\big(\mathrm{PVF}_{i-1} \,\|\, \mathrm{AS}_i \,\|\, \mathrm{seq}\big),$$

where PVF_{i-1} is the field value entering the AS, AS_i is the AS identifier, and seq is a sequence number for replay prevention. Because the PRF output is unpredictable without $K_{s,i}$, any AS that was not on the intended path cannot compute a valid PVF update. Any attempt to insert, reorder, or skip an AS produces a PVF that fails verification at the destination.

Incremental validation. A key feature of L-SPV is that each AS can verify the PVF value entering from the previous hop, catching upstream deviations before forwarding rather than waiting for the destination to verify. This early detection property reduces the propagation of attack traffic through the network.

Comparison to Hermes. L-SPV is the closest prior work to Hermes structurally: both use a single fixed-size in-header field updated at each hop using a shared secret, and both rely on the impossibility of forging the field without that secret. The key differences are: (i) L-SPV uses a standard PRF (e.g. AES-CMAC), which requires modular arithmetic not natively available in P4; Hermes uses the XOR/AND-based accumulator, which is directly expressible in the data plane. (ii) L-SPV is designed for the AS-level Internet with pairwise source-AS keys established during path setup; Hermes uses a pooled DH exchange between the controller and each switch, supporting key rotation without per-flow setup. (iii) L-SPV does not address the opcode obfuscation or LFSR key-reuse mechanisms that Hermes introduces to resist traffic analysis.

3.2.3. Proof-of-Provenance

Wong et al. introduce a *Proof-of-Provenance* protocol [30] for lightweight misrouting detection. The protocol targets scenarios where a sender wants to verify that their packet was forwarded by the intended sequence of ASes, without the overhead of full path-token distribution.

Mechanism. Each AS on the path computes a keyed hash over the current provenance field and its own AS identifier and appends the result to a growing provenance field. The destination checks whether the field corresponds to the expected sequence of hashes. Because the hash is keyed, intermediate ASes cannot forge the contribution of an AS they have not visited, and missing contributions are detectable. The protocol adds only a small constant number of bytes per hop and avoids any public-key operations.

Comparison to Hermes. The emphasis on minimal overhead directly informs one of Hermes's key design principles. However, appending per-hop fields means the header overhead grows with path length at $O(L)$. Hermes takes the overhead argument to its logical conclusion by collapsing all per-hop contributions into a single 32-bit accumulator field, achieving $O(1)$ overhead regardless of the number of hops. The trade-off is that Hermes's accumulator does not allow per-hop incremental verification (each switch cannot check its upstream neighbour's contribution), a property that Wong et al.'s scheme and L-SPV both support.

3.2.4. PFA-INT

Papadopoulos et al. present *PFA-INT* [24], which combines in-band telemetry with per-flow aggregation to reduce the overhead of continuous path monitoring. Standard INT appends per-hop metadata for every packet, producing significant header overhead at high traffic rates. PFA-INT addresses this by aggregating telemetry records across flows at intermediate nodes, emitting summarised reports to a collector only when aggregate metrics change beyond a threshold.

Mechanism. The aggregation is implemented in the P4 data plane using match-action tables keyed on flow identifiers and time windows. Each switch maintains a running aggregate (minimum, maximum, or sum) of selected metrics per flow and emits a digest to the controller only when the aggregate changes significantly. This reduces monitoring traffic by orders of magnitude for stable flows.

Comparison to Hermes. PFA-INT does not provide cryptographic path guarantees: it is a performance monitoring tool rather than a security protocol. An adversary can silently drop or modify telemetry records without detection because the aggregation is unauthenticated. Hermes addresses this gap precisely: by using keyed accumulator operations rather than plain telemetry appends, it ensures that the final accumulated value cannot be forged without knowledge of the per-switch secrets. Where PFA-INT reduces overhead by aggregating across flows, Hermes reduces overhead by collapsing per-hop contributions into a single field — both achieve $O(1)$ per-packet overhead, but Hermes's approach also provides the cryptographic binding that PFA-INT omits.

3.2.5. InvisiFlow

Li et al. present *InvisiFlow* [18] at NSDI 2025, enabling source-controlled routing with strong *path privacy* guarantees. Where most prior work focuses on verifying that a packet took a specific path, InvisiFlow focuses on hiding the path from intermediate nodes during transmission.

Mechanism. InvisiFlow uses a layered onion-encryption scheme adapted for stateless per-hop processing. The source encrypts the packet's routing directives in nested layers, one per hop. Each switch decrypts only its own layer using a pre-distributed key, learning its next-hop forwarding decision but nothing about the remaining path. The encryption is constructed so that each switch's layer is computationally indistinguishable from random bytes to any observer without the per-hop decryption key, preventing traffic analysis of routing decisions. The scheme is stateless at each hop — switches do not maintain per-flow state — keeping the data-plane implementation simple and scalable.

Comparison to Hermes. InvisiFlow and Hermes address complementary security properties. InvisiFlow provides *pre-transmission path privacy*: the path is hidden from observers during forwarding. Hermes provides *post-transmission path integrity*: the Hermes server can confirm the intended path was followed after the fact. Neither property subsumes the other. A combined system applying InvisiFlow's onion layer for privacy and Hermes's accumulator for post-transmission verification would provide both properties simultaneously, at the cost of additional header overhead. The stateless per-hop design of

InvisiFlow is also architecturally compatible with Hermes's design, since both process packets without maintaining per-flow state in the data plane.

3.3. Security in Programmable Data Planes

3.3.1. In-band Network Telemetry (INT)

Kim et al. introduce INT [15], a framework for embedding per-hop network metadata directly into data-plane packets. Each INT-capable switch inserts a telemetry record into a variable-length header stack containing fields such as switch identifier, ingress and egress port identifiers, queue occupancy, and processing timestamps. The records are concatenated at each hop, so the final packet carries a full per-hop trace of its network journey.

Implementation. INT is implemented entirely in the P4 match-action pipeline, with each switch programmed to inspect an *INT shim header* that signals whether telemetry insertion is requested, and to append its own record to the growing stack if so. A *sink switch* or the destination host extracts the accumulated stack and forwards it to a monitoring backend. The design achieves microsecond telemetry latency without measurable throughput degradation, and scales to arbitrary path lengths because switches only process their own record.

Comparison to Hermes. The structural pattern of INT — each hop appends to an in-packet header field, and the accumulated result is read at the end — directly inspired Hermes's accumulator design. Hermes replaces the telemetry metadata stack with a single 32-bit cryptographic accumulator: instead of appending switch identifiers, each switch applies a keyed operation to the field. The critical advantage of this substitution is that the cryptographic operation cannot be forged without the switch's secret key, whereas INT's telemetry fields can be trivially modified by any on-path attacker. Hermes thus inherits INT's efficient in-header accumulation pattern while adding the forgery resistance that INT intentionally omits.

3.3.2. dh-aes-p4

Oliveira et al. present *dh-aes-p4* [23], the first implementation of Diffie–Hellman key exchange with AES encryption running entirely within a P4 programmable data plane, without any SDN controller involvement in the key-exchange process itself.

DH key exchange in P4. The system uses the XOR-based optical DH variant [13] — the same variant adopted by Hermes — replacing modular exponentiation with AND and XOR operations directly expressible in the P4 match-action pipeline. The exchange proceeds as a three-way handshake embedded in special key-exchange packets. Switch s_1 generates a random private value A using the P4 `random()` function and computes its public key $K_A = (G \oplus P) \,\&\, A$. This is sent to s_2, which generates its own private B, computes $K_B = (G \oplus P) \,\&\, B$ and returns it; both switches then independently compute $S = (K_{\text{other}} \,\&\, \text{own}) \oplus P$. Importantly, the private keys are generated on-device by the P4 program and never written to any external register or table, keeping them within the switch's hardware memory.

AES encryption. The shared secret S is used as an AES key to encrypt subsequent traffic. Because P4 does not natively support AES round operations, the authors implement AES via *scrambled lookup tables*: the SubBytes, ShiftRows, and MixColumns transformations are pre-computed into eight tables of 256 × 4-byte entries each (totalling 8 KB of table storage). Each AES round reduces to four table lookups and XOR operations, which are directly expressible as P4 match-action table applications. The round keys are computed offline and installed by a Python controller script, avoiding a compilation bottleneck that would otherwise exceed 30 hours.

Performance. The authors report secret-key renewal RTTs of approximately 0.8–1.0 ms on BMv2 and AES-128/192/256 encryption overhead negligible compared to an unencrypted baseline.

Comparison to Hermes. Hermes directly builds on the DH mechanism from dh-aes-p4, extending it in three ways. First, whereas dh-aes-p4 exchanges one shared secret per pair of switches, Hermes

exchanges a pool of 330–350 keys per switch per session, enabling LFSR-driven key rotation without repeated handshakes. Second, whereas dh-aes-p4 uses the shared key for symmetric payload encryption, Hermes uses its keys for a path-verification accumulator that does not touch payload content. Third, Hermes adds a central verification server, the opcode obfuscation system, and the LFSR key-reuse schedule, none of which have counterparts in dh-aes-p4.

3.3.3. OAT: Attested Execution in the Data Plane

Chen et al. present *OAT* [3], which addresses a different but complementary threat to Hermes: verifying that a P4 switch is actually executing its installed program correctly, rather than a modified version substituted by a compromised vendor or administrator.

Mechanism. OAT runs the P4 program inside a *trusted execution environment* (TEE) on the switch's control plane. The TEE produces a remote attestation report — a cryptographically signed summary of the program binary and its current runtime state — that a verifier can check to confirm the switch is executing an expected, unmodified program. OAT extends this to the data-plane pipeline by instrumenting the P4 program to emit per-packet execution traces, which are aggregated and signed by the TEE and forwarded to the verifier on request. The overhead is kept low by aggregating traces into Merkle-tree-based digests rather than forwarding per-packet records.

Comparison to Hermes. OAT and Hermes address orthogonal threat dimensions. Hermes's threat model assumes that switches execute their P4 program faithfully but that an on-path adversary may manipulate packet contents or bypass a switch entirely. OAT assumes packet contents are correct but that the switch itself may be compromised and running tampered software. Combining the two would provide both *execution integrity* (the switch runs the correct program, via OAT) and *path integrity* (the correct switches were traversed, via Hermes), covering a broader threat surface than either system alone.

3.3.4. P4-IPsec and P4-MACsec

Hauser et al. implement IPsec [9] and MACsec [10] entirely in P4, demonstrating that standardised, complex cryptographic protocols can be expressed in a programmable data plane and deployed on commodity BMv2 switches.

P4-IPsec. The implementation supports both Authentication Header (AH) and Encapsulating Security Payload (ESP) modes, providing packet authentication and payload encryption respectively. AES-GCM encryption and SHA-256 HMAC are implemented using the same scrambled lookup-table technique as dh-aes-p4. Tunnel mode is supported, enabling site-to-site VPN functionality at the switch level. The SDN controller handles key agreement via IKEv2 out-of-band and installs the resulting session keys into the switch's match-action tables.

P4-MACsec. MACsec provides link-layer encryption and authentication between pairs of switches using AES-GCM. A dynamic topology monitoring module detects link changes and triggers automatic key renegotiation, making the system self-healing under topology changes.

Comparison to Hermes. Both P4-IPsec and P4-MACsec confirm the feasibility of P4 as a platform for security protocols, supporting Hermes's choice of P4 as its implementation vehicle. However, both proposals delegate key agreement to an SDN controller running IKEv2 or similar out-of-band. Hermes integrates key exchange directly between the controller and the Hermes server using the optical DH mechanism, avoiding a separate key-management infrastructure. Additionally, P4-IPsec and P4-MACsec provide *payload encryption* between *pairs* of switches on individual links, whereas Hermes provides *path-level verification* across a sequence of switches without modifying the payload at all.

3.4. Performance-Driven Routing

Valancius et al. demonstrate wide-area route control for distributed services [29]. Their system, *VISOR*, allows service providers to express routing preferences to ISPs via a programmatic API; ISPs install corresponding BGP policies to honour those preferences. The system dynamically reoptimises routing

in response to measured network conditions, achieving near-optimal latency and throughput without requiring changes to the BGP protocol itself.

While VISOR is not security-focused, it reinforces a key assumption underlying this thesis: that customisable, application-aware path selection is practically feasible within the existing Internet infrastructure. Hermes extends the notion of performance-driven routing to *security-driven* routing: instead of selecting paths for low latency, the controller selects paths through trusted switches and cryptographically verifies that the selection was honoured.

3.5. Summary and Positioning

Table 3.1 summarises the key properties of the most relevant prior systems and positions Hermes among them across five dimensions: whether the system provides user-level path control, whether it produces post-transmission cryptographic evidence of traversal, whether it includes replay protection, whether it has a data-plane (P4 or equivalent) implementation, and whether it is lightweight enough for high-speed forwarding without heavy per-packet cryptographic operations.

Table 3.1: Comparison of path verification and routing control mechanisms.

System	Path control	Post-facto	Replay protection	P4/data	Lightweight forwarding
ICING [22]	✓	✓	—	—	—
L-SPV [16]	—	✓	✓	—	✓
SCION [25]	✓	✓	—	—	✓
InvisiFlow [18]	✓	—	—	✓	✓
dh-aes-p4 [23]	—	—	—	✓	✓
OAT [3]	—	✓	—	✓	—
UCIP [14]	✓	✓	—	—	—
MCR [20]	✓	✓	—	—	—
Hermes (this work)	✓	✓	✓	✓	✓

The table reveals a clear gap in prior literature that Hermes addresses. ICING and L-SPV both provide post-transmission verification, but their cryptographic primitives (MAC chains using AES or standard PRFs) are not directly expressible in the P4 match-action pipeline without expensive extern functions, and neither has a P4 implementation. SCION provides strong path control and cryptographic path validation at the inter-domain level, using packet-carried forwarding state and per-hop MACs, but it is a clean-slate network architecture rather than an incrementally deployable P4 data-plane implementation. InvisiFlow and dh-aes-p4 both have P4 implementations, but InvisiFlow provides path *privacy* rather than path *verification*, and dh-aes-p4 provides payload encryption between switch pairs rather than multi-switch path proof. OAT addresses execution integrity rather than path integrity. UCIP and MCR provide path control and verification at AS granularity without data-plane implementations.

Hermes is, to the best of our knowledge, the only system in this comparison that simultaneously provides user-controlled forwarding at switch granularity, post-transmission cryptographic path verification, replay protection via sequence numbers and timestamps, and a complete programmable-data-plane implementation evaluated on a Tofino target.

4

Hermes Design

This chapter describes the design of Hermes. Section 4.1 defines the requirements that shape the system. Section 4.2 then gives a high-level overview of the architecture, the actors involved, and the end-to-end packet workflow. The remaining sections describe the threat model, the key-provisioning protocol, the accumulator update rule, the key lifecycle, replay protection, and the packet header formats.

4.1. Design Goals and Requirements

Hermes targets the gap between path control and path verification. A sender may want a packet to traverse a specific sequence of switches or administrative domains, but without cryptographic evidence it cannot later check whether that path was actually followed. The following requirements guide the design.

R1 — Post-transmission path verification. After a packet has been forwarded, Hermes must be able to verify that the packet traversed the selected sequence of Hermes switches. A missing, extra, or incorrect hop should cause verification to fail.

R2 — Constant per-packet overhead. The verification evidence carried in the packet must remain fixed in size, independent of the number of switches on the path. Hermes should therefore have $O(1)$ packet-header growth rather than the $O(L)$ growth of schemes that append one proof or telemetry record per hop.

R3 — Data-plane deployability. The per-hop operation must be expressible directly inside a programmable-switch data plane, rather than offloaded to a control-plane path, so that path verification does not reduce forwarding throughput. This matters because a fixed-function P4 match-action pipeline can only execute simple per-packet arithmetic, bitwise operations, and table lookups at line rate; it cannot natively execute heavyweight primitives such as AES, HMAC, or modular exponentiation without dedicated hardware acceleration or an unacceptable throughput penalty. The per-hop operation must therefore be restricted to primitives a P4 pipeline can execute natively.

R4 — Resistance to bounded on-path observation. An adversary observing packet contents on the links, including accumulator values entering and leaving a switch, should not be able to forge valid evidence or recover switch secret material before the relevant key material is rotated or renegotiated.

R5 — Replay resistance. An adversary must not be able to capture a packet that verified successfully and replay it later to obtain a second false confirmation for a path it did not legitimately traverse.

4.2. Overview

Hermes is a path-verification system for programmable data planes. It lets a trusted controller steer data packets through a selected sequence of Hermes switches and then cryptographically confirm, after transmission, that the packet traversed those switches in the correct order (R1).

For a path of length L, the sender receives an ordered switch sequence $(s_1, \ldots, s_L)$ and a per-flow nonce from the Hermes server.[1] Every data packet carries a small, fixed-size accumulator field in the Hermes header, so the verification evidence does not grow with path length (R2). When the packet reaches switch s_i, the switch applies a keyed operation to the accumulator using secret material shared only with the Hermes server. The packet then continues to the next hop with the updated accumulator. Crucially, the operation applied at each hop is itself kept secret: the switch and the Hermes server agree not only on a key but also on which of several possible operations that key triggers. An on-path observer can see the accumulator enter and leave a switch, but without knowing which operation was applied it cannot reproduce the transformation or forge a valid update, which is what gives Hermes its resistance to on-path eavesdroppers (R4).

After the final Hermes switch s_L, the packet reaches the receiver, which reports the final accumulated value to the Hermes server. The server independently reconstructs the same computation using its copy of the per-switch keys, opcodes, and key schedules. If the recomputed value matches the reported accumulator, the server returns Accept; otherwise it returns Reject. Because all per-hop evidence is folded into one accumulator field, the packet overhead remains constant in the path length. A per-flow sequence number and timestamp window travel alongside the accumulator so that a captured packet cannot be replayed for a second false Accept (R5); these mechanisms are described in Section 4.6.

At a high level, the design separates data-plane and control-plane responsibilities. The data plane only performs lightweight fixed-width arithmetic and bitwise operations on packet headers, which keeps the per-hop operation deployable directly on programmable switch hardware (R3). The control plane handles key provisioning, mTLS-protected communication with the Hermes server, digest handling, and server-side verification. This separation allows Hermes to keep the forwarding path simple while still producing cryptographic evidence about the path a packet followed.

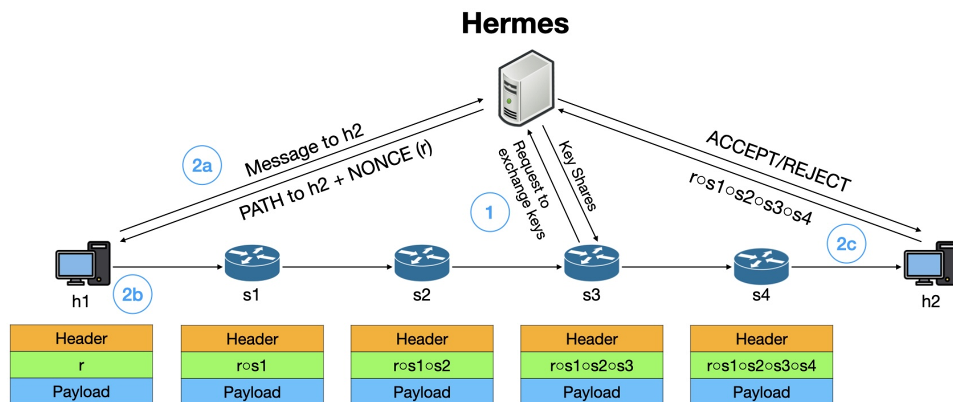


Figure 4.1: High-level Hermes architecture, illustrated for an example path of length $L = 4$ (generically $s_1, \ldots, s_L$, with sender h_1 and receiver h_2).

[1] How the Hermes server selects or computes this path — for example from routing policy or operator configuration — is outside the scope of this thesis; Hermes takes the path as given and verifies that packets complied with it. An equally valid alternative would let the sender propose its own path and have the Hermes server simply issue a nonce for it; the verification mechanism described below is unaffected by which side chooses the path.

4.2.1. Actors and Components

Hermes consists of three main actor types.

Hermes server. The Hermes server is the trust anchor of the system. It stores the cryptographic state for each Hermes switch: key pools, opcode assignments, LFSR seeds, and the server-side copy of each switch's key schedule. It also issues the selected path and per-flow nonce to the sender, and answers verification requests by replaying the expected accumulator computation.

Hermes switch actor. A Hermes switch actor consists of a programmable data-plane switch and an associated control-plane component. The control-plane component provisions the switch with key material and opcode assignments, communicates with the Hermes server, and relays digests emitted by the data plane. The data-plane switch processes each Hermes data packet by looking up its current key and opcode, updating the accumulator, advancing its key schedule when required, and forwarding the packet or emitting a digest if it is the final Hermes hop. Independently of its position on the path, any switch may also emit a local digest to its own control-plane component to signal that its key pool is nearing exhaustion; Section 4.7 unifies this with the final-hop digest format.

Each switch is trusted to execute its assigned P4 program and control-plane logic correctly. However, switches are treated as independent: a switch only learns its own key pool, opcode assignments, and LFSR seed. It does not learn the secret material of other switches on the path.

Sender and receiver. The sender obtains a selected path and per-flow nonce from the Hermes server. It then originates data packets carrying the Hermes headers. The receiver is the packet destination and is responsible for initiating verification in the current design: after receiving a Hermes packet, it extracts the final accumulator, hop count, sequence number, timestamp, nonce, and flow identifier from the Hermes header and sends these values to the Hermes server in a `VERIFY` request. The Hermes server then checks the reported values by replaying the expected accumulator computation for the selected path.

Deployment granularity. The unit of verification in Hermes is a Hermes switch actor, but the granularity at which that actor is deployed is a deployment choice. A Hermes switch actor could be every switch within an Autonomous System (AS), giving switch-level verification throughout the domain, or it could be a single border switch per AS, in which case Hermes verifies AS-level traversal rather than the path inside each AS. Hermes' verification mechanism is unchanged either way; what changes is which physical switches run the Hermes data plane. Inter-domain policy negotiation and agreement on such paths are outside the scope of this thesis.

4.2.2. Operational Phases

Hermes operates in three phases.

Key provisioning. Before traffic is verified, each Hermes switch actor establishes a pool of shared keys, one opcode per key, and an LFSR seed with the Hermes server (step 1 in Figure 4.1). These values are installed into the switch's match-action tables and registers.

Session setup and operation. For each flow, the sender requests a path from the Hermes server and receives the ordered switch sequence and per-flow nonce (step 2a in Figure 4.1). It then sends Hermes data packets along the selected path (step 2b). Each switch updates the accumulator, and the receiver extracts the final Hermes header and emits a message for server-side verification.

Key rotation and renegotiation. During operation, each switch advances through its own key pool independently of every other switch on the path. It first uses keys sequentially and then reuses keys according to an LFSR-driven schedule. When a switch's own key pool nears the point where reuse would expose the system to repeated-observation attacks, that switch, whichever position it occupies on the path, signals its control plane directly to perform a fresh key exchange. This phase supports R4 by bounding the number of observations under reused key material (step 2c).

4.2.3. Per-Packet Verification Workflow

Once the switches are provisioned and a sender has obtained a path and nonce, every data packet follows the same workflow.

1. **Origination.** The sender emits a packet carrying a Hermes data header. The accumulator is initialised from the per-flow nonce issued by the Hermes server. The header also carries the flow identifier, sequence number, timestamp, and `hop_count`. The `hop_count` starts at zero.
2. **Per-hop accumulation.** At switch s_i, the P4 program reads the switch's current key index from internal register state, looks up the corresponding key and opcode, applies the keyed operation to the accumulator, increments the `hop_count`, and forwards the packet according to the flow-specific forwarding state.
3. **Verification request.** After the packet has traversed the selected switch sequence $(s_1, \ldots, s_L)$, it reaches the final destination. The receiver extracts the final Hermes header values and sends a `VERIFY` request to the Hermes server. This request contains the flow identifier, reported accumulator, hop count, sequence number, timestamp, and nonce.
4. **Server-side replay.** The Hermes server replays the accumulator computation from the same per-flow nonce, applying its stored keys and opcodes for the selected switch sequence $(s_1, \ldots, s_L)$. The server reconstructs each switch's key index by replaying its copy of the switch's key schedule; the key index is therefore never carried in the packet.
5. **Verdict.** If the reported accumulator matches the server's recomputed value, the server returns Accept; otherwise it returns Reject.

4.3. Key Provisioning: Optical Diffie–Hellman

Before forwarding begins, each Hermes switch actor establishes cryptographic material with the Hermes server. Hermes uses the optical Diffie–Hellman-style construction introduced in Section 2.3.4, which derives a batch of shared secret values using only XOR and AND operations rather than modular exponentiation. This bitwise-only structure is what makes a DH-style exchange feasible on devices that cannot evaluate modular exponentiation, such as P4 switches. In this prototype the exchange is still carried out by the switch control-plane component and the Hermes server rather than inside the P4 data plane itself, but the same property is what would allow the exchange to be pushed into the data plane in a future deployment.

All provisioning and verification messages are exchanged over a mutually authenticated TLS (mTLS) channel. This includes `DH_INIT`, `DH_RESP`, `DH_KEY`, `DH_KEY_RESP`, and `VERIFY` messages. This protection is important because the raw optical-DH transcript should not be treated as confidential when sent in clear. mTLS provides confidentiality, integrity, and endpoint authentication for the control-plane exchange, while the P4 data plane remains limited to lightweight accumulator operations.

Setup phase. The control-plane component for switch s_i generates a batch of $N_k \in [330, 350]$ secret values B_j and computes public values

$$K_{B,j} = (G \oplus P) \,\&\, B_j$$

for each $j \in \{0, \ldots, N_k - 1\}$. It then sends a `DH_INIT` message to the Hermes server containing s_i, the public parameters G and P, the key count N_k, and the list $K_{B,0}, \ldots, K_{B,N_k-1}$.

The Hermes server:

1. generates N_k secret values A_j and computes $K_{A,j} = (G \oplus P) \,\&\, A_j$;
2. computes shared values $S_j = (K_{B,j} \,\&\, A_j) \oplus P$;
3. assigns a random 3-bit opcode op_j to each key;
4. generates a non-zero 32-bit LFSR seed σ;
5. replies with `DH_RESP` containing N_k, $K_{A,0}, \ldots, K_{A,N_k-1}$, $op_0, \ldots, op_{N_k-1}$, and σ.

The switch control plane computes the same shared values locally as

$$S_j = (K_{A,j} \,\&\, B_j) \oplus P$$

and installs the resulting keys and opcodes into the switch's match-action tables. The Hermes server retains its own copy of the same material.

Choice of $N_k \in [330, 350]$. The key-pool size is drawn freshly from the interval $[330, 350]$ rather than fixed. Randomising N_k prevents an observer who counts packets from knowing exactly when a switch transitions from the sequential phase to the LFSR-driven reuse phase. The upper end of the range is also chosen so that a `DH_INIT` message carrying N_k public values fits inside a single 1448-byte TCP/IP payload, avoiding IP fragmentation of the provisioning exchange: at 4 bytes per public value, 350 keys occupy approximately 1400 bytes, leaving headroom for the remaining `DH_INIT` fields within that budget.

Key regeneration. Each switch tracks its own key index and packet counter locally, independently of the other switches on the path. When a switch nears the point where key reuse would become unsafe, that switch sets a `trigger_dh` flag in its own digest, regardless of whether it is the final hop. The control-plane component then performs a `DH_KEY` exchange, identical in structure to `DH_INIT`, to obtain a fresh key pool, opcode assignments, and LFSR seed. The fresh material is then installed into the switch tables and registers.

4.4. Accumulator Update Rule

The central per-packet operation is:

$$acc_{\text{next}} = apply_op(op_I,\ k_I,\ acc_{\text{current}})$$

where I is the current key index, k_I is the key at that index, and op_I is the opcode assigned to that key.

4.4.1. Operations

Eight operations are defined, each identified by a 3-bit opcode, as listed in Table 4.1.

Table 4.1: Accumulator update operations and their 3-bit opcodes.

Opcode (bin)	Name	Expression
000	Add	$acc + k$
001	Rot(b rot a)	$rotl(k,\ acc)$
010	Rot(a rot b)	$rotl(acc,\ k)$
011	And	$acc \,\&\, k$
100	Or	$acc \mid k$
101	Sub(b-a)	$(k - acc) \bmod 2^{32}$
110	Sub(a-b)	$(acc - k) \bmod 2^{32}$
111	Xor	$acc \oplus k$

4.4.2. Rotation Operation

The rotation operation $rotl(a, b)$ is defined as follows:

1. Extract the rotation amount: $r = (b \gg 27) \,\&\,$ `0x1F`, giving $r \in \{0, \ldots, 31\}$.
2. Extract the addend: $d = b \,\&\,$ `0x07FFFFFF`.
3. Rotate a left by r bits: $a' = (a \ll r) \mid (a \gg (32 - r))$.
4. Return $a' + d \pmod{2^{32}}$.

This operation is non-commutative: $rotl(acc, k) \neq rotl(k, acc)$ in general. It therefore provides two distinct opcode behaviours from one basic primitive.

4.4.3. Security Rationale

If Hermes used one fixed operation for every key, an observer could often infer the key from the accumulator before and after a switch. For example, under addition the key would be $k = acc_{\text{out}} - acc_{\text{in}} \pmod{2^{32}}$. Hermes therefore assigns a secret opcode to each key. For a single observation, the adversary does not know which of the eight operations produced the observed output, which increases the uncertainty of the transformation. This does not by itself remove all risk under repeated key reuse; that risk is bounded through key rotation and renegotiation and is analysed in Chapter 5.

4.5. Key Lifecycle

Each switch maintains internal state for its key schedule:

- the current key index I;
- the current LFSR state, which is zero during the sequential phase;
- a packet counter used to decide when to advance to the next key.

After every `refresh_rate` packets, the key advances.

Sequential phase. The switch initially consumes its key pool in order. Each time the refresh interval elapses, the key index advances by one:

$$I \leftarrow I + 1.$$

This continues until the initial pool has been consumed.

LFSR phase. After the sequential phase, the switch enters LFSR-driven reuse. The LFSR state is seeded with σ, received during key provisioning, and stepped as:

$$\ell_{\text{next}} = \begin{cases} (\ell \gg 1) \oplus \texttt{0xB4BCD35C} & \text{if } \ell \mathbin{\&} 1 = 1, \\ \ell \gg 1 & \text{otherwise.} \end{cases}$$

The next key index is derived from the LFSR state and mapped into the key-pool range. Because the sequence is determined by the seed σ, both the switch and the Hermes server can reconstruct it deterministically, while an observer without σ cannot predict the reuse order directly.

The key index is not transmitted in the packet. Each switch reads its current key index from internal state, and the Hermes server reconstructs the corresponding index by replaying its copy of the switch's key schedule.

Renegotiation. Each switch on the path monitors its own key index against the limit analysed in Chapter 5 independently of every other switch. When a given switch's own reuse would approach that limit, it requests fresh key material by setting the `trigger_dh` flag in a digest it emits to its own control-plane component; this is independent of whether that switch is the final hop on the path. The control plane then performs a new mTLS-protected `DH_KEY` exchange and reinstalls that switch's key pool and opcode table.

4.6. Replay Protection

Replay protection addresses the case where an adversary captures a packet that already verified successfully and resubmits it later. Hermes uses a sequence number and timestamp window to reject such attempts, while the nonce binds packets to the server-issued flow session.

Sequence number. Each packet carries a per-flow monotonically increasing sequence number `seq` (32 bits). The Hermes server records accepted $(flow_id, seq)$ pairs and rejects a repeated pair.

Timestamp window. Each packet carries a millisecond Unix timestamp `timestamp_ms` (48 bits). The server rejects a verification request whose timestamp falls outside a ±30 s window of its own clock. This bounds replay state and prevents old captures from being reused much later. The width of the window

trades off two concerns: it must comfortably exceed the clock skew and queueing delay expected between sender and server so that legitimate packets are not rejected, while remaining small enough to bound the server's per-flow replay-state memory to a manageable interval.

Nonce. Each flow carries a 32-bit nonce issued by the Hermes server during session setup. The nonce is the same for every packet of a flow. It does not itself block replay, because a replayed packet carries the same nonce, but it binds the packet stream to a server-issued session and seeds the initial accumulator.

Together, the sequence number and timestamp window support Requirement R5, while the nonce provides flow binding and accumulator initialisation.

4.7. Packet Header Format

Hermes packets carry a small base header followed by a type-specific payload. For data packets, the payload is the Hermes data header containing the accumulator and verification fields.

4.7.1. Message Types

The base header identifies the message type. Only `MSG_DATA` packets traverse the data plane. Control-plane messages such as verification requests and key-provisioning messages are exchanged between the controller or switch control-plane component and the Hermes server over the mTLS-protected control channel.

Table 4.2: Hermes message types.

Type ID	Name	Direction / purpose
0	`MSG_DATA`	Sender → receiver, processed by each Hermes switch
1	`MSG_VERIFY`	Control plane → Hermes server: verify accumulator
2	`MSG_RESPONSE`	Hermes server → control plane: ACCEPT/REJECT
3	`MSG_DH_REQ`	Control plane → Hermes server: establish or refresh key pool
4	`MSG_DH_RESP`	Hermes server → control plane: keys, opcodes, and LFSR seed

4.7.2. Data Packet Header

The Hermes data header carries the fields listed in Table 4.3. The `key_index` is deliberately not carried in the packet; it is internal switch state and is reconstructed by the server.

Table 4.3: Fields of the Hermes data header.

Field	Width	Description
`accumulator`	32 bits	Cryptographic accumulator; updated at each hop
`hop_count`	16 bits	Number of Hermes hops completed; serves as hop index
`flow_id`	32 bits	Identifies the flow/path session
`seq`	32 bits	Per-flow monotonically increasing sequence number
`timestamp_ms`	48 bits	Sender timestamp in milliseconds
`nonce`	32 bits	Per-flow nonce/seed issued by the Hermes server
Total	**24 bytes**	

Unlike an IP TTL, `hop_count` is not decremented to limit packet lifetime. It is incremented by each Hermes switch and records how many Hermes hops have already processed the packet.

Together with the 4-byte Hermes base header (`msg_type`, `version`, and `length`), a Hermes data packet therefore adds 28 bytes of protocol overhead.

4.7.3. Digest to the Control Plane

Hermes uses a single digest format, listed in Table 4.4, that any Hermes switch may emit to its own control-plane component; how the control plane handles a digest depends on which fields are populated. The final receiver on a path emits a digest restating the final accumulator and anti-replay fields, which its control-plane component relays to the Hermes server as a `VERIFY` request (Section 4.2.3). Independently of path position, any switch — not only the final hop — may also emit a digest with the `trigger_dh` flag set to report that its own key pool is nearing exhaustion; when `trigger_dh` is set, the receiving control-plane component handles the digest locally by initiating a `DH_KEY` renegotiation (Section 4.3) rather than relaying it to the Hermes server.

Table 4.4: Fields of the Hermes digest format. The final hop populates all fields, including the verification-relevant accumulator and anti-replay fields; any switch signalling local key-pool exhaustion sets `trigger_dh` and `hop_idx`.

Field	Width	Description
`flow_id`	32 bits	Flow identifier
`hop_count`	16 bits	Number of Hermes hops completed
`accumulator`	32 bits	Final accumulated value
`seq`	32 bits	Sequence number from the data packet
`timestamp_ms`	48 bits	Timestamp from the data packet
`nonce`	32 bits	Nonce from the data packet
`trigger_dh`	1 bit	Set when fresh key material should be provisioned
`hop_idx`	32 bits	Index of the switch that requested renegotiation

5

Security Analysis

This chapter analyses the security of Hermes against the threat model of Section 2.4. The aim is not a formal proof in the sense of a MAC-based path-validation protocol, but to establish what the accumulator construction protects against, where its security margin comes from, and where its genuine limitations lie.

Summary of findings. The headline result is that successful undetected tampering is highly unlikely within the lifetime of a key generation. Forging a single hop that a packet did not traverse succeeds with probability about 1/8, even if the adversary has recovered the key value. Forging a whole L-hop path with about $(1/8)^L$ — already small for any realistic path, and that is even after recovering all of the keys. But the decisive barrier is not this per-packet ambiguity: it is that *hijacking* a hop requires the adversary to predict the *next* transformation a switch will apply, and that prediction requires reconstructing the entire randomised secret schedule. Each switch draws its key *and* its operation from two independently shuffled secret arrays, indexed by a hidden LFSR sequence whose state is never transmitted. To anticipate the next key an adversary must therefore recover (i) the secret ordering of the key array, (ii) the independently secret ordering of the opcode array, and (iii) the LFSR seed and its current phase. Our cryptanalysis (Section 5.3) shows this needs on the order of 10^3 observed flows through a single switch merely to cover the index space, *plus* a brute-force search over the 2^{32} LFSR seed space, because the index indirection removes the algebraic shortcut a plain LFSR would offer. Renegotiation discards and replaces the whole schedule long before any of this can complete, so the adversary's partial knowledge is reset before it is ever useful. The one limitation that genuinely matters is infrastructural rather than cryptographic: the key-provisioning exchange must run over a confidential, authenticated channel, because its transcript otherwise reveals the shared secret and bypasses everything above.

5.1. Security Goals and Threat Model

Recall from Section 2.4 that the adversary is an on-path man-in-the-middle. It can read and modify packet fields, inject, drop, delay, reorder, and replay packets, and may observe the accumulator entering and leaving a switch. It cannot read switch or server memory, and it cannot directly obtain switch keys, opcode assignments, or LFSR seeds.

Against this adversary, Hermes targets three properties:

G1 — Path-forgery resistance. A packet that did not traverse the authorised path should not yield an accumulator the server accepts. This covers skipped, inserted, and reordered hops, and manually corrupted accumulators.

G2 — Unpredictability of future transformations. Observing accumulator input/output pairs should not let the adversary predict what a switch will do to the *next* packet. This — not the recovery of a single past key value — is the data-plane security question that matters, because only prediction of the next transformation lets an adversary bypass a hop undetected.

G3 — Replay and provisioning security. A previously accepted packet must not be accepted again, and the key-provisioning channel must not reveal the key array, the opcode array, or the LFSR seed.

Table 5.1 summarises the outcomes established in the rest of the chapter.

Table 5.1: Security properties of Hermes against the defined threat model.

Threat	Defence / mechanism	Outcome
Single-hop forge, one observation	Opcode ambiguity (8 hidden operations)	$\approx 1/8$
Full-path forge	Independent per-hop secrets	$\approx (1/8)^L$
Missing, extra, or reordered hop	Server-side accumulator replay	Reject
Predict the *next* key (hijack)	Hidden index + two shuffled arrays + LFSR seed	Infeasible before renegotiation
Replay	Sequence number + timestamp window	Reject
Key-exchange eavesdropping	Confidential, authenticated channel (mutual TLS)	Mitigated
Unauthorised control-plane client	Client-certificate authentication	Reject

5.2. Forgery Resistance

For the server to return Accept, the reported accumulator must equal the value obtained by replaying the authorised sequence of switch transformations. For a path of length L,

$$A_{\text{final}} = f_{s_L}\left(\cdots f_{s_1}(v)\right), \qquad f_{s_i}(x) = apply_op(op_{I_i}, k_{I_i}, x),$$

where v is the per-flow nonce (the accumulator seed), k_{I_i} is the key switch s_i currently uses, and op_{I_i} is the secret opcode bound to that key. The server holds the same arrays and schedules, so it recomputes this value independently. Any skipped, inserted, reordered, or modified hop changes the applied sequence, and the final accumulator no longer matches.

An adversary with no observations and no key material can only guess the 32-bit final value, succeeding with probability 2^{-32} per attempt. A stronger adversary observes a switch transform one packet and then tries to forge a later one. Given a single pair $(A_{\text{in}}, A_{\text{out}})$, it can invert the transformation under each of the eight opcode hypotheses, producing the candidate sets of Table 5.2.

Table 5.2: Candidate keys implied by one observation, per opcode hypothesis.

Opcode	Forward operation	Candidates	Type
Add	$a + k$	1	Unique inversion
Sub(a-b)	$a - k$	1	Unique inversion
Sub(b-a)	$k - a$	1	Unique inversion
Xor	$a \oplus k$	1	Unique inversion
Rot(b rot a)	$rotl(k, a)$	1	Unique inversion
Rot(a rot b)	$rotl(a, k)$	0–32	One per valid rotation amount
Or	$a \mid k$	$2^{\text{popcount}(A_{\text{in}})}$	Set-valued
And	$a \,\&\, k$	$2^{32-\text{popcount}(A_{\text{in}})}$	Set-valued

Because the opcode is itself secret and selected from the eight possibilities, a single observation leaves an average per-hop forgery probability of about

$$P^{(1)}_{\text{forge}} \approx \frac{1}{8}.$$

This is an average, not a uniform guarantee: uniquely invertible opcodes leave few consistent hypotheses, whereas And/Or leave many. Figure 5.1 shows the resulting mixture of hypothesis counts, which

averages to a single-guess success near 0.125. For a full path of L independently keyed switches, an adversary with one useful observation per switch succeeds with about $(1/8)^L$; a four-hop path is already $\approx 1/4096$. This bound holds *as long as the relevant keys have not been recovered* — the subject of the next section.

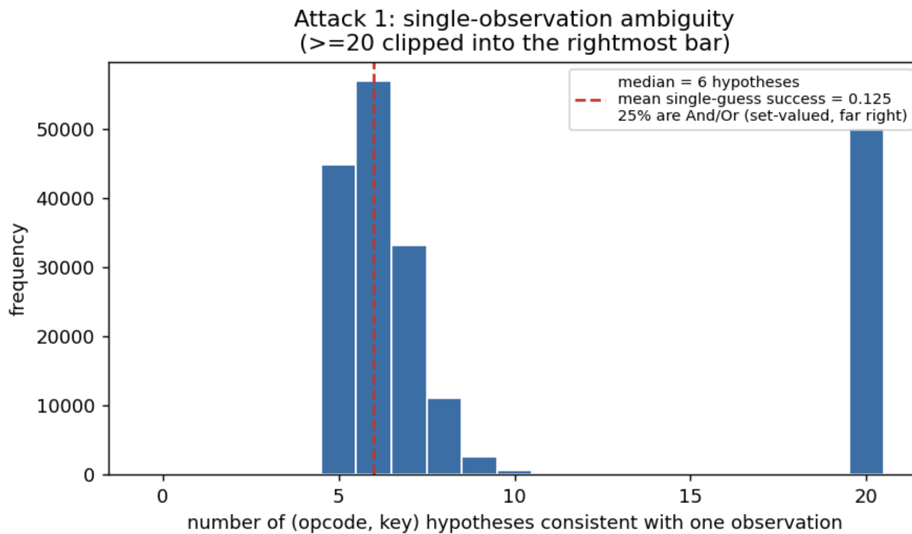


Figure 5.1: Number of (op, k) hypotheses consistent with one observation (≥ 20 clipped into the rightmost bar). The mixture averages to a single-guess forgery success of approximately 0.125.

5.3. Why Key Recovery Does Not Enable Hijacking

The natural worry is that an adversary who watches a switch long enough recovers its keys and then impersonates it. This section shows why that worry does not translate into a practical attack on Hermes. The crucial distinction is between recovering the *value* of a key that was used in the past and predicting the *next* transformation a switch will apply. Only the latter enables a hop to be bypassed undetected, and the latter is the hard problem.

What an adversary can and cannot learn from reuse. Hermes does not reuse a key within a safe window during normal operation; the question here is the worst case, in which an adversary accumulates many observations of one switch across many flows and asks what it can extract. For a uniquely invertible opcode such as Xor, inverting two observations of the *same* key under the correct opcode yields the same value ($k = A_{\text{in}} \oplus A_{\text{out}}$), whereas a wrong opcode almost never produces a repeat. In principle, then, recurring candidate values betray reused keys, and an adversary could recover a set of key *values* by frequency analysis. The decisive point is that this is not enough. The adversary cannot even tell *which* of its observations used the same key, because the key index is never transmitted and the key order is randomised; and recovering a key's value does not reveal its *position* in the schedule, which is what determines when (if ever) it is used again. Knowing a bag of past key values is therefore useless for predicting the next one.

Why prediction needs the whole secret schedule. To anticipate the transformation a switch applies to the next packet, an adversary must reproduce three independently hidden quantities at once: the next key index the LFSR will emit, the key value stored at that index, and the opcode stored at that index in a *separately* shuffled opcode array. The two arrays are permuted independently, so even perfect knowledge of every key value does not reveal which opcode accompanies it, and neither array's ordering is observable from the wire. Reconstructing the orderings is a double coupon-collector problem: the adversary must observe and disambiguate every index of both arrays, which takes on the order of $N_k \ln N_k$ observations of a single switch and, with the per-observation 1/8 ambiguity and value collisions, substantially more in practice.

The LFSR seed has no shortcut. A textbook LFSR is linear and can be solved from a handful of output bits. Hermes defeats this by never exposing the LFSR output: the bits are consumed as *indices* into the secret arrays rather than combined into the accumulator, so there is no system of linear equations to solve. Recovering the seed therefore reduces to brute force over the 2^{32} seed space ($\approx 4.3 \times 10^9$ candidates), each tested by replaying the full schedule. We confirmed this empirically: a linear (GF(2)) seed-recovery attack that succeeds against a plain LFSR fails entirely against the index-driven schedule.

Putting the numbers together. Figure 5.2 reports an end-to-end experiment in a deliberately weakened setting — a reduced 16-bit LFSR, and with the key *values* already granted to the adversary — so that only the schedule and array orderings remain to be learned. Even then, the hijack-ready capability (correctly predicting the next hop) stays low until the adversary has observed roughly a thousand flows through one switch, and approaches certainty only after two to three thousand, governed by how long it takes to cover both secret arrays. In the real system the 16-bit seed becomes a 32-bit seed, adding the 2^{32} brute-force factor on top. The combined cost — on the order of 10^3 observed flows for array coverage *multiplied by* a 2^{32} seed search — is what an adversary must pay before it can forge even one future hop, and renegotiation replaces the entire schedule well within the reuse phase, i.e. long before that coverage is reached. The practical defence is therefore simply to renegotiate before the reuse phase runs long enough to expose the arrays; the key-pool size affects only *when* reuse begins (the birthday bound below) and is a secondary knob, not the defence.

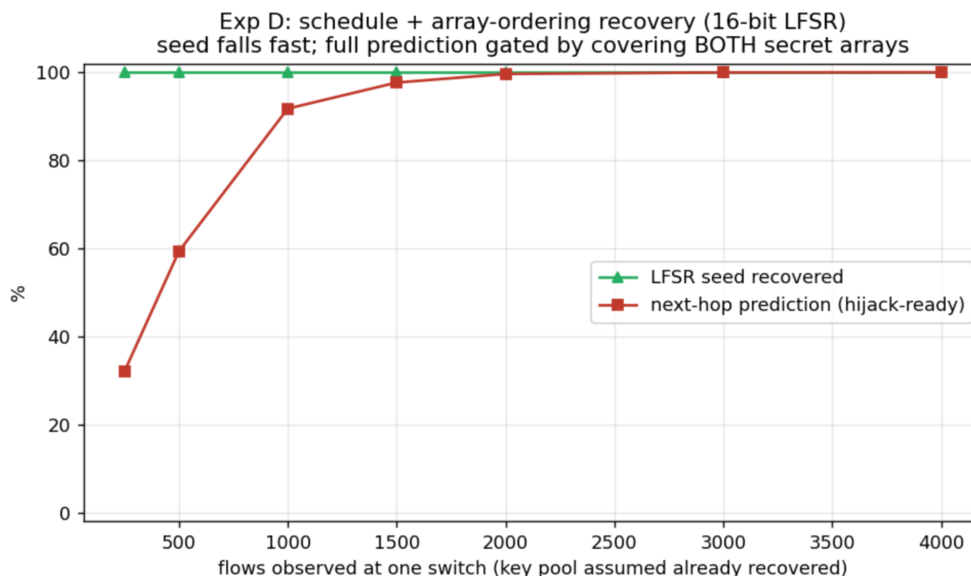


Figure 5.2: Schedule and array-ordering recovery in a weakened 16-bit-LFSR model with key values already granted to the adversary. The LFSR seed itself falls quickly once the model is reduced, but full next-hop prediction (the hijack-ready capability) is gated by covering *both* independently shuffled secret arrays, and only approaches certainty after ~2,000–3,000 observed flows at one switch. At the real 32-bit seed a 2^{32} search is required in addition.

The birthday bound, and why it is a side issue. The *birthday bound* is the observation that among N equally likely values, a repeated draw (a "collision") is expected after only about $\sqrt{N}$ draws rather than N — the same reason a shared birthday is likely in a room of about 23 people when $N = 365$. Applied to a key pool of size N_k, the expected number of draws before some index recurs is

$$E[T_{\text{revisit}}] \approx \sqrt{\frac{\pi N_k}{2}},$$

about 23 for $N_k = 340$. This tells us only when key *reuse* starts, not when an attack succeeds. As argued above, reuse on its own leaks nothing the adversary can act on, because it cannot identify which observations share a key and cannot map values to indices. The birthday bound is thus a useful guide for *when to renegotiate* — comfortably before the reuse phase exposes the arrays — rather than a vulnerability in itself.

5.4. Replay Resistance

Replay protection stops an adversary re-submitting a packet that previously verified. Hermes relies on three fields. The sequence number is the actual blocker: the server stores accepted ($flow_id, seq$) pairs and rejects repeats, so a verbatim replay fails outright. The timestamp bounds the server's state and the attack window — packets outside the configured ±30 s window are rejected, so old captures cannot be replayed later and the server need not remember all historical sequence numbers. The nonce binds a packet to its server-issued flow and seeds the accumulator, but does not itself prevent replay, since a replayed packet carries the same nonce.

Under the assumption that the server keeps replay state within the timestamp window, verbatim replay is rejected with probability 1. A modified replay must still produce a valid accumulator and therefore reduces to the forgery problem of Section 5.2.

5.5. Conclusion

Hermes detects any single-packet path deviation with overwhelming probability, and the capability needed to bypass a hop undetected — predicting a switch's next transformation — is computationally infeasible within the lifetime of a key generation, because it demands the joint recovery of two independently shuffled secret arrays and a 2^{32} LFSR seed that the index indirection leaves with no algebraic shortcut. Renegotiation resets this entire state well before an adversary can assemble it. The residual risk is not in the data plane but in provisioning: the key-exchange transcript must remain confidential and authenticated, since its disclosure hands the adversary the secret state directly. Securing that channel (Section 2.4) is therefore the precondition on which the rest of the analysis depends.

6

Evaluation Methodology and Results

This chapter evaluates Hermes on a hardware testbed. For each experiment, we first state the *goal*, then describe the setup and procedure, and finally interpret the results. Unless stated otherwise, every measurement was collected on the testbed described in Section 6.2. Results derived from static analysis rather than a live run are marked as such.

A recurring theme is that Hermes' latency is dominated by communication between its components — switch, controller, and verification server — rather than by packet processing inside the Tofino pipeline. We therefore characterise these paths explicitly in Section 6.2 and interpret latency results against the plain-forwarding baseline defined there.

6.1. Research Questions

RQ1 Does the system correctly detect the defined path-deviation attacks?

RQ2 What is the per-switch Diffie–Hellman key-exchange latency, and how does it scale with the number of keys exchanged?

RQ3 What is the end-to-end verification latency from packet injection to Accept/Reject, how is it split across components, and how does it behave as the offered probe rate grows?

RQ4 What forwarding throughput does the Hermes data plane sustain, and where is the bottleneck relative to plain forwarding?

RQ5 What overhead does key rotation and renegotiation add to steady-state operation?

RQ6 What fixed header overhead does Hermes add per packet?

RQ7 What flow-completion time does Hermes deliver across a heavy-tailed flow-size distribution?

6.2. Experimental Setup

Environment. All hardware experiments run on a single Intel Tofino switch (`sw1`) programmed with the TNA build of the Hermes P4 data plane. Two hosts are attached in the line topology

$$h_1 \to s_1 \to h_2,$$

where s_1 is the only physical Hermes hop in the hardware testbed.[1] Host h_1 injects Hermes data frames, the switch updates the accumulator, and h_2 receives the forwarded frames.

The Hermes server and the controller run on the switch host. The controller programs the data plane over BF-RT (Barefoot Runtime) gRPC, the gRPC-based control-plane API used to install and update Tofino match-action tables and registers, and reaches the Hermes server over a mutually authenticated TLS (mTLS) connection on the loopback interface. Earlier development versions of the prototype used software P4 tooling, but the measurements reported in this chapter are based on the Tofino hardware

[1] The implementation artefacts used in this thesis are available at: `https://github.com/CallumH2410/Hermes`.

implementation. Because the forwarding table matches Hermes-specific fields, stock TCP/UDP load generators cannot be used directly. Instead, the experiments use a purpose-built generator that emits frames with the same Hermes header layout as the controller-generated probe packets.

The single-switch testbed evaluates one physical Hermes hop. Multi-hop behaviour in the path-deviation experiment is emulated through recirculation on the same switch; this is stated again in Experiment 1. Table 6.1 lists the hardware and software versions used throughout.

Table 6.1: Hardware and software environment.

Component	Version / Details
Switch	Intel Tofino (`sw1`)
SDE	Intel P4 Studio SDE 9.13.2
P4 architecture	Tofino Native Architecture (TNA)
P4 compiler	`bf-p4c` (P4Studio)
Control plane	BF-RT gRPC to Tofino; mTLS socket to server
mTLS library	OpenSSL (controller–server mutual TLS)
Operating system	Ubuntu (switch host)
Python	3.8 (SDE `tofino` site-packages)
C++ compiler	GCC, `-O2 -std=c++17`

Parameters fixed across all experiments. The following parameters restate definitions already introduced in Chapter 4; they are repeated here as a single reference point for reproducing the experiments, and any experiment-specific deviation is called out explicitly in that experiment.

- Public DH generators: G = `0x5A5A5A5A`, P = `0xA5A5A5A5`
- Keys per switch per DH exchange: $N_k \in [330, 350]$ (drawn each exchange)
- LFSR polynomial: Galois 32-bit, mask `0xB4BCD35C`
- Replay window: ±30 s timestamp, per-flow sequence number
- `flow_id`: 1 (single installed path)

Inter-component RTTs and the plain-forwarding baseline. Hermes spreads verification across three components: the switch, the controller, and the Hermes server. In this testbed, no artificial inter-component delay is configured. The software components are co-located on the switch host, the controller–server channel is an OpenSSL mTLS session over loopback, the controller–switch channel is the BF-RT gRPC stream, and the host–switch links are direct cable attachments. The reported latencies therefore represent a minimum-RTT case. A distributed deployment with a remote verifier would add network RTT to the verification component.

We compare latency and throughput results against a plain-forwarding baseline: the same physical path $h_1 \rightarrow s_1 \rightarrow h_2$ running a minimal forwarding pipeline without Hermes accumulation, digest emission, or verification. This baseline isolates the cost of the testbed itself from the cost introduced by Hermes. No artificial delay is emulated between h_1 and s_1 or between the switch host and either endpoint; the baseline therefore reports the testbed's own minimum achievable round trip rather than a delay we configured.

The baseline forwarding round trip was measured by reflecting frames back to h_1 and timestamping send and return on one clock over 2,000 samples. The median forwarding RTT is 0.095 ms, with a minimum of 0.083 ms, p95 of 0.114 ms, maximum of 0.136 ms, and standard deviation of 0.008 ms. This corresponds to a one-way forwarding latency of roughly 0.05 ms. Against this baseline, the verification latency reported in Experiment 3 is dominated by digest delivery and server-side verification rather than ordinary data-plane forwarding.

Statistical methodology. Latency experiments are repeated for 30 runs per configuration and reported using the median, minimum, maximum, and 95th percentile. One warm-up run is discarded per batch

to reduce cold-start connection-setup and server-state effects. Throughput is reported over fixed 10 s generation windows per frame size. Flow-completion time is reported as percentiles over 3,000 flows in a single run. Python timing uses `time.perf_counter()`, and capture timestamps use the kernel/libpcap clock.

6.3. Experiment 1: Path-Deviation Detection

Goal. The functional goal of Hermes is to accept packets that followed the authorised path and reject packets that deviate from it. This experiment checks both directions: no false rejects for the correct path and no missed detections for the defined deviation cases.

Setup and procedure. We inject 30 packets for each scenario and record the verdict. The multi-hop scenarios (missing and extra hop) require more than one Hermes hop and are therefore emulated through recirculation on `sw1`; the single-hop replay and timestamp scenarios run directly. The scenarios are:

1. correct path;
2. missing hop;
3. extra rogue hop applying an arbitrary accumulator update;
4. replayed packet;
5. stale timestamp more than 30 s old;
6. corrupted accumulator before the final hop.

Results. Table 6.2 shows that Hermes classified all 180 trials correctly. It accepted all correct-path packets and rejected every deviation using the intended mechanism. This confirms functional path-integrity behaviour under the assumption that per-switch keys and opcodes remain secret; the conditions under which that assumption can fail are analysed in Chapter 5. *Answers* ***RQ1***.

Table 6.2: Path-deviation detection (30 trials each). All cases classified correctly.

Scenario	Expected	Mechanism	Detected
Correct path (baseline)	Accept	—	30/30
Missing hop	Reject	Accumulator mismatch	30/30
Extra hop inserted	Reject	Accumulator mismatch	30/30
Replayed packet	Reject	Sequence-number check	30/30
Stale timestamp	Reject	Timestamp window	30/30
Corrupted accumulator	Reject	Accumulator mismatch	30/30

6.4. Experiment 2: DH Key-Exchange Latency

Goal. A switch joins Hermes by completing a key-provisioning exchange before it can verify traffic, and it repeats the exchange during renegotiation. The goal is to measure whether this cost is small and whether it remains stable as the number of keys per exchange changes.

Setup and procedure. We performed the `DH_INIT`/`DH_RESP` exchange for s_1 thirty times at each of $N_k \in \{330, 340, 350\}$ keys. Timing starts when the controller sends `DH_INIT` and ends when it has received and parsed `DH_RESP`. The exchange runs over the loopback mTLS socket and does not involve the data plane, so this isolates key-provisioning latency from switching overhead.

Results. Figure 6.1 shows that the exchange RTT is effectively constant across the tested range, with a median of 1.84–1.89 ms and only a 0.05 ms increase over twenty additional keys. A 350-key message is approximately 6% larger than a 330-key message, yet the median grows by less than 3%, which indicates that the exchange is dominated by round-trip and parsing cost rather than by payload size. The wider

whisker at $N_k = 330$ reflects occasional scheduler jitter. At approximately 1.85 ms, a single exchange is small relative to the renegotiation interval. *Answers **RQ2**.*

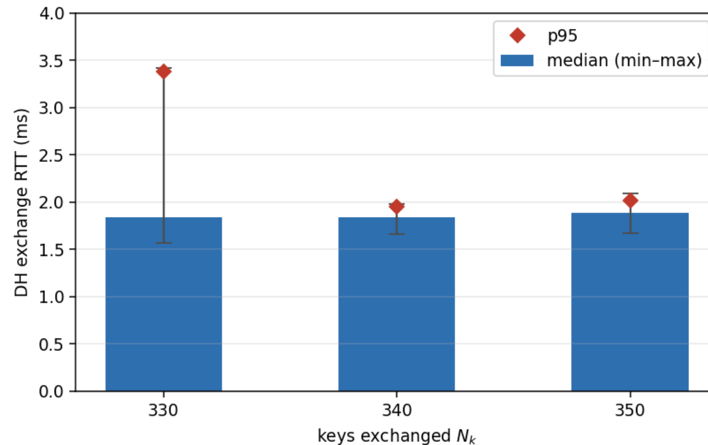


Figure 6.1: DH key-exchange RTT versus key count on `sw1` (30 runs each): median bar, min–max whisker, and p95 marker.

6.5. Experiment 3: End-to-End Verification Latency

Goal. Hermes should return an Accept/Reject verdict with low and consistent latency. This experiment measures the additional latency introduced by digest delivery and server-side verification, and determines whether that latency changes as the offered probe rate increases.

Setup and procedure. We injected Hermes probe packets at four offered rates: 1, 10, 50, and 100 pps. Each rate is measured over 30 packets, with a fresh Hermes server instance per rate to reset replay state. For each packet we record:

- digest latency t_d: packet injection to controller receipt of the switch digest;
- verify latency t_v: digest receipt to Accept returned by the Hermes server;
- total latency $t_{\text{total}} = t_d + t_v$.

These probe rates are intentionally low in bandwidth terms: even at 100 pps and 1500-byte frames, the offered load is only about 1.2 Mbit/s. This experiment is therefore not a throughput stress test; it isolates digest-delivery and verification latency. Throughput is evaluated separately in Experiment 4.

Results. Figure 6.2 shows that the median total latency remains within 5.76–5.92 ms across the tested 1–100 pps range. This 0.16 ms spread indicates that verification latency is effectively rate-independent at these probe rates. The digest path contributes approximately 1.9 ms, while the verify path contributes approximately 3.9 ms and dominates the total.

The verify component is dominated by the controller opening a fresh mTLS connection per request. A persistent pooled mTLS connection would remove the per-request handshake and is expected to reduce t_v. Compared with the plain-forwarding baseline of 0.095 ms RTT, the Hermes verification path is about 80× the bare forwarding latency. This confirms that Hermes' measured cost is in the control-plane verification path, not in ordinary data-plane forwarding. All 120 packets were accepted, confirming correct operation at every tested rate. *Answers **RQ3**.*

6.6. Experiment 4: Data-Plane Forwarding Throughput

Goal. Hermes updates the accumulator on every packet, so this experiment checks whether that per-packet operation reduces forwarding throughput relative to plain forwarding.

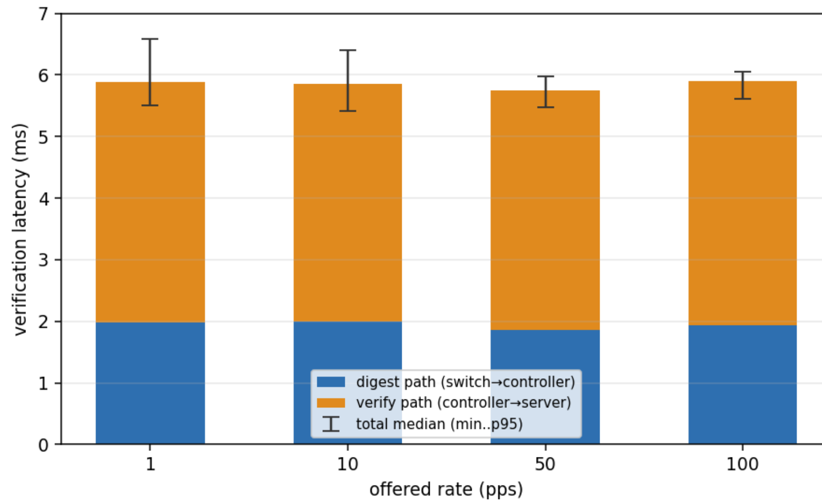


Figure 6.2: End-to-end verification latency by offered probe rate, split into digest latency and server-side verify latency.

Figure 6.3: Plain-forwarding baseline versus Hermes verification latency. The baseline forwarding RTT has median 0.095 ms, while the Hermes end-to-end verification median is approximately 5.88 ms.

Setup and procedure. We drive `flow_id=1` from h_1 for fixed 10 s windows at six on-wire frame sizes from 64 to 1500 bytes, using a single packet-generation socket. For each frame size we record the achieved send rate in packets per second and offered throughput in Gbps. Delivery is cross-checked against the lossless capture used in Experiment 7, and the result is compared against the plain-forwarding baseline.

Results. Figure 6.4 compares Hermes with plain forwarding. Both packet-rate series are essentially flat at approximately 0.38–0.47 Mpps regardless of frame size. This flatness indicates a fixed per-packet cost in the software sender rather than a switch limit. Offered throughput rises with frame size, from approximately 0.2 Gbps at 64-byte frames to approximately 4.7 Gbps at 1500-byte frames.

Hermes and the plain-forwarding baseline track each other closely. At the offered rate, Hermes therefore adds no measurable forwarding-throughput penalty relative to plain forwarding. The flow-completion capture in Experiment 7 records all 3,249,157 transmitted packets with zero loss, confirming that the switch forwards the offered load without drops.

Limitations. Because both Hermes and the baseline are limited by the software generator, this result establishes parity at the offered rate, not the absolute switch ceiling. Measuring the true line-rate ceiling requires a stronger generator such as `pktgen`, TRex, or MoonGen, together with working switch-side TX counters.

6.7. Experiment 5: Key Rotation Overhead

Goal. Hermes rotates keys during normal operation. The common key-advance events should add negligible overhead, while the less frequent full renegotiation may be more expensive. This experiment measures both cases.

Setup and procedure. We run packets continuously through s_1 with `refresh_rate` configured to force key advances at known points. We measure the additional per-packet latency at three events relative to a packet that triggers no key advance: a sequential advance, an LFSR-driven advance, and a full DH renegotiation that performs a `DH_KEY` exchange and table reinstall.

Results. Figure 6.5 shows that sequential and LFSR advances add a median of 0.06–0.07 ms and remain below 0.2 ms at p95. These common events execute inside the switch pipeline as register reads and

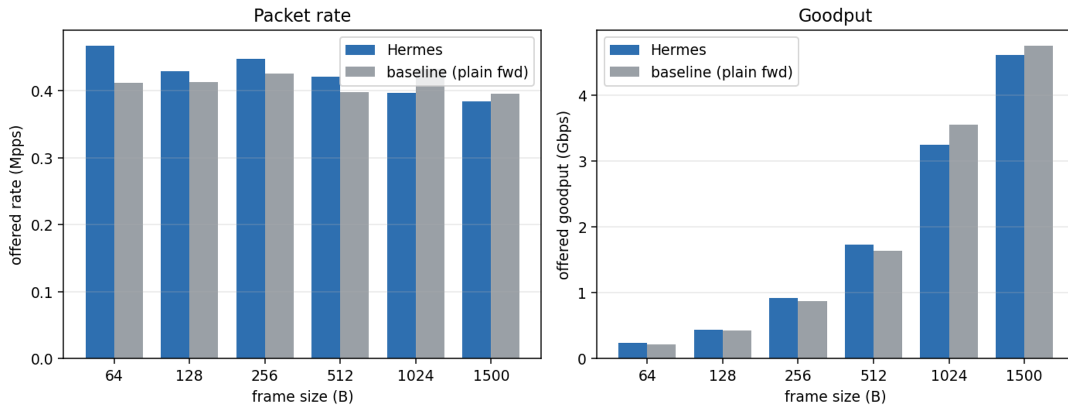


Figure 6.4: Offered forwarding throughput over 10 s windows, Hermes versus the plain-forwarding baseline. Packet rate is flat for both, indicating a software-generator bottleneck.

writes and do not require a control-plane round trip. Full DH renegotiation is more expensive, with a median of 7.31 ms, because it requires an mTLS-protected `DH_KEY` exchange and table reinstall. Since renegotiation occurs only after many packets, its amortised contribution to steady-state latency is small. *Answers* ***RQ5****.*

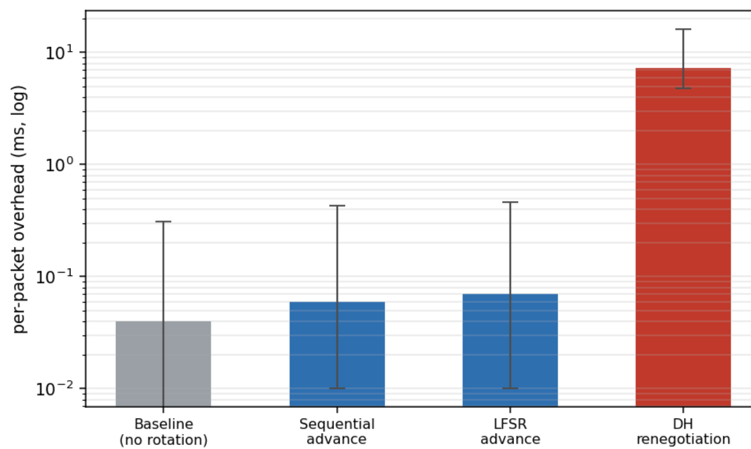


Figure 6.5: Per-packet key-rotation overhead relative to a no-rotation baseline. Sequential and LFSR advances are pipeline-only; full renegotiation is control-plane dominated.

6.8. Experiment 6: Header Overhead

Goal. A path-verification scheme should add a small, fixed amount of packet overhead regardless of path length. This experiment computes Hermes' per-packet header overhead and compares it with hop-proportional alternatives.

Setup and procedure. We compute the overhead from the Hermes header definitions and confirm it against generated Hermes data frames and live captures. This is a static analysis, so it is exact rather than measured over repeated runs.

Results. Table 6.3 lists the fields. The Hermes base header contributes 4 bytes, and the Hermes data header contributes 24 bytes, for a total of 28 bytes per data packet. This overhead is fixed and independent of the number of switches on the path, so Hermes has $O(1)$ header growth.

For a 1500-byte Ethernet frame, the 28-byte Hermes header corresponds to approximately 1.87% of the frame. Relative to a 1448-byte payload budget, it corresponds to approximately 1.93%. This is the

packet-space cost of carrying the Hermes verification state: it reduces the usable payload per packet by that fraction, but the throughput experiment does not show a measurable forwarding penalty at the offered rate.

This compares favourably with hop-proportional schemes. ICING carries one token per hop (≈20–32 B each, or 60–96 B for three hops), while INT adds $L \times r$ bytes (r ≈12–24 B per hop, or 36–72 B for three hops) without a cryptographic path-verification guarantee. Hermes instead keeps the verification state fixed at 28 bytes, independent of the path length. Figure 6.6 illustrates this directly: Hermes' overhead is flat in the number of hops L, while ICING and INT both grow linearly with L, so the gap in Hermes' favour widens as the path gets longer. *Answers **RQ6**.*

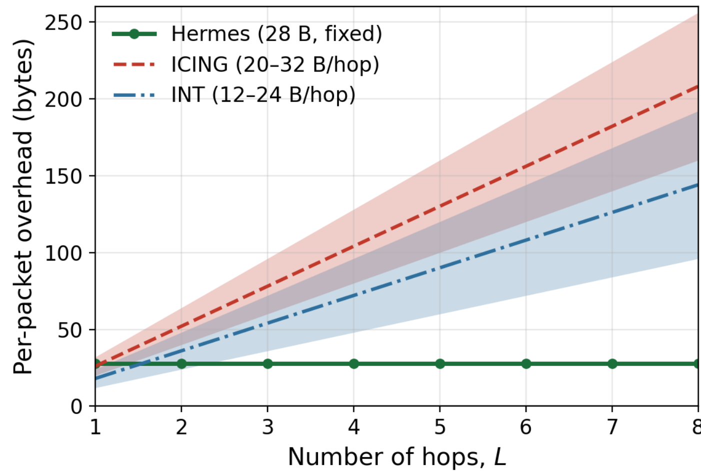


Figure 6.6: Per-packet header overhead versus path length L. Hermes adds a fixed 28 bytes regardless of L ($O(1)$), while ICING and INT add overhead that grows linearly with L ($O(L)$); shaded bands show the per-hop cost ranges given in the text.

Table 6.3: Per-packet header overhead introduced by Hermes.

Header field	Size (B)	Cumulative (B)
`hermes_base_t`: `msg_type`	1	1
`hermes_base_t`: `version`	1	2
`hermes_base_t`: `length`	2	4
`hermes_data_t`: `accumulator`	4	8
`hermes_data_t`: `hop_count`	2	10
`hermes_data_t`: `flow_id`	4	14
`hermes_data_t`: `seq`	4	18
`hermes_data_t`: `timestamp_ms`	6	24
`hermes_data_t`: `nonce`	4	28
Total Hermes overhead	**28**	**28**

6.9. Experiment 7: Flow Completion Time

Goal. Beyond per-packet latency, we want to observe how Hermes behaves across complete flows under a heavy-tailed flow-size mix. The purpose is to check whether packets are delivered losslessly and whether receiver-observed completion span grows predictably with flow size.

Setup and procedure. We generate 3,000 flows from h_1 using a heavy-tailed flow-size distribution and Poisson arrivals. Each B-byte flow is sent as $\lceil B/1500 \rceil$ Hermes frames, with each frame tagged so arrivals can be grouped by logical flow. Arrivals at h_2 are captured losslessly with `tcpdump` and converted for analysis.

We report flow completion time as the receiver-observed completion span:

$$\mathrm{FCT} = t^{\mathrm{rx}}_{\mathrm{last}} - t^{\mathrm{rx}}_{\mathrm{first}},$$

measured on the single receiver capture clock, rather than the conventional definition of FCT as the time from the sender issuing the first packet of a flow to the sender receiving the final acknowledgment for it. We deviate from the conventional definition because sender and receiver timestamps were taken on independent, unsynchronised clocks in this run, which makes the absolute send-to-receive time unrecoverable; the receiver span is the sound metric available given that constraint. Its limitations are that single-packet flows have a zero span and that the constant edge latency common to all flows is omitted.

Results. The capture was complete: all 3,000 flows and all 3,249,157 packets arrived with zero loss (Table 6.4). Figure 6.7 shows median and p99 receiver span increasing proportionally with flow size above approximately 10 kB, as expected when completion is transmission-dominated. The median rises from 0.14 ms in the 10–100 kB band to 36 ms above 5 MB. Below 10 kB, the span sits at the timing floor because such flows consist of only a few packets.

The CDF in Figure 6.8 shows an overall median of 0.28 ms, dominated by small flows, and a p99 of 88.5 ms, set by the large-flow tail. For flows above 1 MB, the received goodput clusters around a median of 1.87 Gbps, reflecting the software generator's burst rate rather than a switch bottleneck. The zero-loss capture confirms that the switch keeps up with the offered workload.

A direct plain-forwarding FCT baseline with the same flow-size seed was not collected. Therefore, this experiment should be interpreted as a workload sanity check and loss test for Hermes under a heavy-tailed flow mix, rather than as a precise measurement of Hermes' incremental flow-completion overhead. *Answers **RQ7**.*

Table 6.4: Workload and capture summary (3,000 flows, single run).

Quantity	Value
Flows generated	3,000
Packets transmitted	3,249,157
Packets captured at h_2	3,249,157 (100%, 0 loss)
Flows fully received	3,000 (100%)
Flow size min / median / mean / max	1.0 kB / 74.6 kB / 1.62 MB / 28.7 MB
Single-packet flows	93 (3.1%)

6.10. Summary

Hermes detects every defined path deviation in the functional tests (**RQ1**), confirming correct operation before turning to performance. On the performance side, Hermes exchanges keys in approximately 1.85 ms independently of key count (**RQ2**), and verifies a packet end-to-end in approximately 5.9 ms, rate-independent over 1–100 pps and dominated by the per-request mTLS verify path rather than data-plane processing (**RQ3**). Measured against a plain-forwarding baseline whose round trip is 0.095 ms, the verification path is about 80× the bare forwarding latency, so the cost is concentrated in the control plane. Hermes forwards the offered workload at the same rate as plain forwarding without loss (**RQ4**). Sequential and LFSR key advances are effectively free, while full DH renegotiation costs approximately 7 ms and occurs infrequently (**RQ5**). Hermes adds a fixed 28-byte header regardless of path length, which compares favourably with the hop-proportional overhead of ICING and INT (**RQ6**). Finally, Hermes delivers receiver-observed flow spans that scale predictably with flow size (**RQ7**).

The remaining measurement gaps are a line-rate generator with working switch-side TX counters for the absolute forwarding ceiling, a persistent server connection to reduce verification latency, host-clock synchronisation for absolute send-to-receive FCT, and a plain-forwarding FCT baseline with the same workload seed.

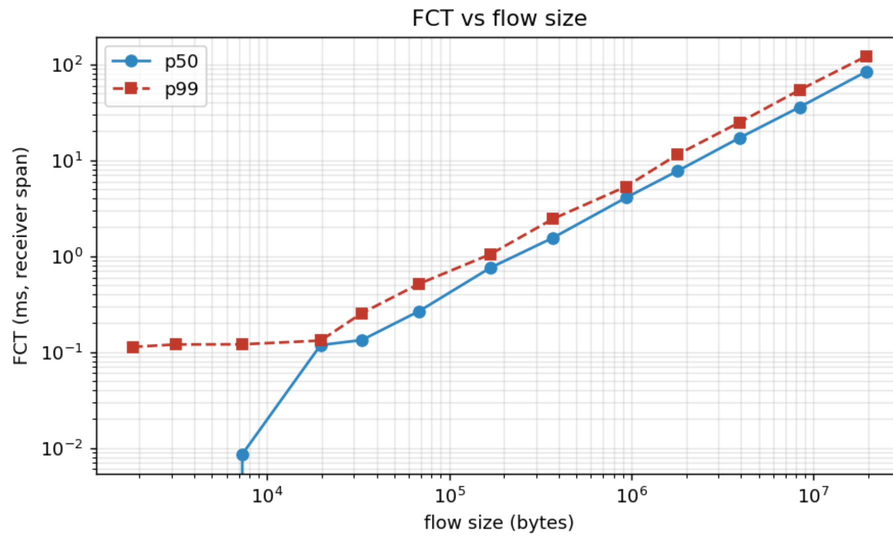


Figure 6.7: Flow completion time as receiver span versus flow size, p50 and p99. Completion scales linearly with size above approximately 10 kB; small flows sit at the timing floor.

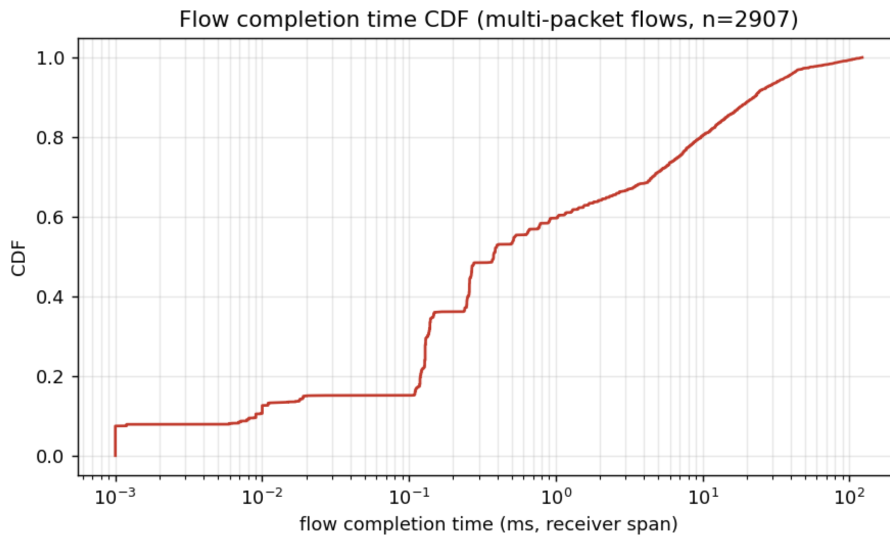


Figure 6.8: Flow completion time CDF as receiver span over multi-packet flows. The heavy tail corresponds to large flows.

7 Discussion

7.1. Interpretation of the Results

The results show that Hermes is technically feasible as a lightweight path-verification prototype for programmable data planes. The most important outcome is that the per-packet data-plane work remains deliberately simple: switches do not compute hashes, signatures, or block-cipher rounds, but instead perform table lookups, register updates, and fixed-width accumulator operations. This is the central design achievement of Hermes. It demonstrates that cryptographic path evidence can be produced inside the constraints of a P4-programmable forwarding pipeline.

The latency results should be interpreted in this context. Hermes does not aim, in its current form, to verify every production packet at arbitrary Internet scale. It is better understood as a mechanism for verified probes, selected security-sensitive flows, or intra-domain accountability experiments. The measured verification latency is dominated by the control-plane path: digest delivery, communication with the Hermes server, and server-side verification. The forwarding operation itself remains lightweight. This distinction is encouraging, because it means the main performance bottlenecks are not in the P4 data-plane operation but in software components that can be optimised. Persistent mTLS connections to the Hermes server, batching of verification requests, and more efficient digest handling would likely reduce end-to-end verification latency further.

The key-exchange results are also encouraging. Hermes exchanges hundreds of keys per switch in one provisioning step, and the measured latency remains stable across the tested key-pool sizes. This suggests that provisioning is not a dominant steady-state overhead. Since full renegotiation is only required when the key lifecycle approaches an unsafe reuse region, its millisecond-scale cost is acceptable for the evaluated use case. The addition of mutual TLS strengthens this part of the design by protecting the provisioning transcript and authenticating the control-plane endpoint, without changing the P4 data-plane mechanism.

The path-deviation results provide the clearest functional validation of the system. Missing-hop, extra-hop, replay, stale-timestamp, and corrupted-accumulator scenarios are all rejected in the experimental setup. These experiments show that the server-side replay logic and switch-side accumulator updates remain synchronised, and that the replay-protection fields work as intended. The experiments cover defined attack cases rather than all possible adversarial strategies, so they should be read together with the security analysis. Nevertheless, they demonstrate that the implemented prototype correctly enforces the path-integrity property it was designed to test.

7.2. Security Trade-offs

Hermes explores a deliberate point in the design space between strong cryptographic path-validation systems and deployable programmable-switch mechanisms. Standard path-validation systems often rely on MACs or PRFs because these primitives provide well-studied unforgeability guarantees. Hermes does not use them in the data plane because their implementation cost conflicts with the design goal

of simple P4-native processing. Instead, Hermes uses a lightweight accumulator construction that is easier to deploy in a programmable switch pipeline, while accepting a smaller security margin than a conventional MAC chain.

The main consequence of this trade-off is the need to control key reuse. A single observation of an accumulator input and output does not generally reveal the key, because the adversary does not know which opcode was used. Repeated observations under the same key can eventually reduce this uncertainty. The key pool, opcode assignments, randomised pool size, and LFSR-driven reuse schedule are therefore best understood as practical mitigation mechanisms: they delay and complicate recovery, and they give the system a defined point at which renegotiation should occur.

The compact accumulator is another conscious design choice. A 32-bit value keeps header overhead low and is attractive for a P4 implementation, but it also limits the collision and guessing resistance of the prototype. For the purpose of this thesis, this is an appropriate trade-off because the goal is to explore feasibility under realistic data-plane constraints. For a production deployment, a wider accumulator or multi-word construction would provide a stronger security margin at the cost of additional header and pipeline resources.

The controller-to-server channel is protected using mutual TLS, so provisioning and verification messages are encrypted and both endpoints authenticate each other. This closes the direct control-plane eavesdropping route against the key pool, opcode assignments, and LFSR seed. The remaining security assumption is operational: private keys and certificates must be managed correctly, and other trusted control-plane channels must receive equivalent protection in a full deployment.

7.3. Scope and Generalisability

Hermes should be interpreted as a switch-level path-verification system, not as a complete replacement for inter-domain routing security architectures. It operates most naturally within a controlled domain where programmable switches can be deployed and a trusted Hermes server can manage keys and verification. This makes data centres, enterprise networks, research networks, regulated backbones, and operator-controlled segments natural early deployment targets.

The design is nevertheless relevant beyond the specific prototype. Many networks already deploy programmable switches, SDN controllers, Segment Routing, or telemetry systems. Hermes shows that such infrastructure can be extended with compact cryptographic path evidence. In these environments, Hermes could allow operators to verify that selected flows traversed approved middleboxes, avoided untrusted segments, or passed through required inspection points. This is a practical and useful form of accountability even before considering full Internet-scale deployment.

The design also has a useful relationship to existing work. Compared with INT, Hermes adds cryptographic binding to per-hop evidence. Compared with ICING or SCION-like MAC chains, Hermes sacrifices some cryptographic strength to gain P4-native deployability and constant-size evidence. Compared with InvisiFlow, Hermes addresses path integrity rather than path privacy. These differences suggest that Hermes is best seen as one practical point in a larger design space rather than as a universal optimum.

7.4. Toward Multi-Domain Hermes Deployments

A real-world deployment would likely require more than one Hermes server. A single server is suitable for the prototype because it gives one trust anchor a complete view of the selected path, the per-switch key material, and the verification state. At larger scale, however, one Hermes server could oversee only a group of Autonomous Systems. Multiple Hermes domains would then be needed, each responsible for verifying the part of the path inside its own administrative region.

In such a model, a Hermes server would act as the verifier for a group of ASes or operator-controlled switches. While a packet remains inside that group, the local Hermes server can verify the accumulator in the same way as in the current design. When the path crosses from an AS belonging to one Hermes group into an AS belonging to another Hermes group, the outgoing Hermes server would verify the accumulator produced so far. If the partial path is accepted, it would communicate an authenticated handoff value to the next Hermes server. That value could serve as the starting accumulator or nonce for

the next segment of the flow, allowing the next Hermes group to continue verification without needing to know the internal key material of the previous group.

This would turn end-to-end verification into a chain of locally verified path segments. Each Hermes server would prove that the packet correctly traversed its own domain, then handoff a verified state to the next domain. The final verdict could be constructed from the sequence of accepted segment proofs. Such a design would preserve administrative separation: no Hermes server would need access to another domain's switch keys, opcode assignments, or LFSR seeds. It would also make Hermes more scalable, because verification state and key management would be distributed across domains.

The main challenge in this multi-server model is trust coordination. Hermes servers would need to authenticate each other, agree on the format and meaning of handoff messages, and possibly reach consensus on the validity and ordering of inter-domain segment proofs. This could be implemented through a federation of Hermes servers, a hierarchical trust model, or a consensus protocol among servers responsible for adjacent domains. Designing such an architecture is outside the scope of the current prototype, but it is a promising direction because it shows how Hermes could evolve from a single-domain verifier into a broader accountability layer for multi-domain routing.

7.5. Limitations

Several limitations remain, but they are best understood as the boundary of the current prototype rather than as failures of the approach. First, Hermes assumes trusted switch execution. If a switch is compromised and deviates from the installed P4 program, Hermes may not detect that the switch misbehaved internally. This limitation is common to many programmable data-plane systems. Combining Hermes with switch attestation mechanisms would strengthen the trust model.

Second, the current accumulator is compact but cryptographically limited. The 32-bit field and eight-operation opcode space are suitable for a feasibility prototype, but a production system would benefit from a stronger construction. A wider accumulator, keyed permutation, or P4-compatible MAC approximation would improve security at the cost of additional header and pipeline resources.

Third, key reuse remains the dominant data-plane security consideration. The LFSR schedule reduces predictability but does not eliminate the fundamental risk that repeated observations under reused keys can reveal information. Future designs should investigate larger key pools, safer renegotiation thresholds, per-flow key derivation, or hybrid constructions that avoid reuse more aggressively.

Fourth, the current system relies on a single Hermes server. This simplifies verification and key management for the prototype, but larger deployments would require a distributed or hierarchical verifier architecture. The multi-domain model discussed above is one possible direction for scaling Hermes beyond a single administrative region.

Fifth, the evaluation is limited in scale. The prototype demonstrates feasibility and correctness under the tested topology, path length, and probe rates, but it does not yet establish behaviour under wide-area delay, many concurrent flows, multiple administrative domains, or adversarial traffic at high volume. These are natural next steps rather than contradictions of the current results.

7.6. Future Work

Several directions follow naturally from this thesis. The first is strengthening the cryptographic construction. A wider accumulator, larger operation space, or P4-compatible keyed mixing function could improve security while retaining deployability. Another direction is to derive per-flow or per-packet keys from a switch-local seed, reducing the risks of direct key reuse.

The second is improving the protected control plane. The current prototype already uses mutual TLS for the controller-to-server channel. A production-quality deployment should extend equivalent authentication and encryption to all relevant control-plane channels, define certificate-management procedures, and use persistent protected connections to avoid repeated handshake overhead.

The third is improving performance. Persistent mTLS connections, verification batching, asynchronous server processing, and more efficient digest handling could reduce the current verification latency. Hardware experiments at higher rates and with longer paths would clarify the true throughput ceiling.

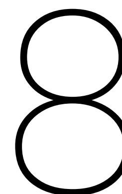
The fourth is integrating Hermes with existing routing mechanisms. Segment Routing, SDN controllers, and INT-style telemetry systems are natural candidates. Hermes could be used to verify that segment lists were actually followed, to authenticate selected telemetry records, or to provide audit evidence for policy-based routing decisions.

The fifth is extending Hermes to multi-domain deployments. A federation of Hermes servers could verify path segments across groups of ASes, exchange authenticated handoff values at domain boundaries, and combine local segment proofs into a broader end-to-end path-verification result. This would require inter-server authentication, handoff semantics, and possibly consensus between Hermes servers, but it offers a realistic path from the current single-server prototype toward wider deployment.

The sixth is broadening the threat model. Combining Hermes with switch attestation would address compromised-switch scenarios. Combining Hermes with path-privacy mechanisms such as onion-style forwarding would provide both confidentiality of the route during forwarding and integrity of the path after delivery.

7.7. Closing Perspective

Hermes demonstrates that controllable and verifiable routing can be explored using today's programmable data-plane tools. Its main contribution is not that it solves Internet path verification completely, but that it shows a concrete, implementable path between two extremes: unauthenticated telemetry on one side and heavyweight clean-slate cryptographic routing architectures on the other. By using a compact accumulator, lightweight per-hop operations, protected control-plane key provisioning, and P4-compatible processing, Hermes provides a practical foundation for further research into accountable, user-controlled packet forwarding.



Conclusion

This thesis investigated whether user-defined packet routing can be combined with efficient, cryptographic post-transmission path verification in programmable data planes. The central challenge was to provide evidence that a packet followed an intended route without adding per-hop header growth or relying on heavyweight cryptographic operations inside the forwarding pipeline.

Hermes addresses this challenge with a compact accumulator-based design. A sender obtains an ordered path and per-flow nonce from the Hermes server. As a packet traverses the selected path, each switch applies a keyed operation to the packet's accumulator. The final accumulator is reported to the server, which independently recomputes the expected value and returns an Accept or Reject verdict. The key idea is that all per-hop evidence is folded into one fixed-size field, allowing Hermes to keep packet overhead constant while still producing verifiable path evidence.

The main conclusion is that this approach is feasible within the constraints of a programmable data plane. Hermes avoids AES, HMAC, modular exponentiation, and other heavyweight primitives in the forwarding path, using only fixed-width arithmetic and bitwise operations that are compatible with P4. The prototype demonstrates fixed per-packet overhead, stable millisecond-scale verification latency, successful mTLS-protected key provisioning, and correct detection of the tested path-deviation and replay scenarios. The forwarding experiments also show that, in the evaluated setup, the observed throughput is limited by the software packet generator rather than by the switch pipeline itself.

The answer to the research question is therefore conditionally affirmative. Hermes shows that efficient post-transmission path verification is possible at switch granularity using programmable data planes. It can steer packets through an intended sequence of switches, accumulate per-hop cryptographic evidence in a compact header field, and verify that evidence at a trusted server. This makes Hermes suitable as a foundation for path-integrity monitoring, verified probes, and accountability mechanisms in controlled programmable-network environments.

The answer is conditional because the security guarantee depends on the stated threat model. Hermes assumes that switches execute their P4 programs correctly, that the Hermes server remains trusted, that mTLS certificates and private keys are managed correctly, and that key renegotiation occurs before repeated observations under reused keys reveal too much information. The current accumulator is deliberately lightweight and P4-compatible, but it does not provide the same security margin as a conventional MAC-based path-validation scheme. These are design trade-offs rather than contradictions: Hermes explores a practical point in the design space between unauthenticated telemetry and heavyweight cryptographic routing architectures.

Overall, Hermes demonstrates that programmable data planes can do more than forward packets or export telemetry. They can participate directly in producing cryptographic evidence about the path a packet followed. While Hermes is not intended as a complete Internet-scale routing architecture, it establishes that lightweight, constant-overhead path verification is achievable within realistic data-plane constraints and provides a basis for future work on stronger accumulators, distributed verification,

switch attestation, and larger-scale deployment.

References

[1] Fred Baker. *RIP Version 2*. RFC 2453. 1998. URL: https://datatracker.ietf.org/doc/html/rfc2453.

[2] Pat Bosshart et al. "P4: Programming Protocol-Independent Packet Processors". In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95. DOI: 10.1145/2656877.2656890.

[3] Zaoxing Chen et al. "OAT: Attested Execution Policies for Switch Data Planes". In: *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2020.

[4] David Cooper et al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280. RFC Editor, May 2008. DOI: 10.17487/RFC5280. URL: https://www.rfc-editor.org/rfc/rfc5280.

[5] Whitfield Diffie and Martin Hellman. "New Directions in Cryptography". In: *IEEE Transactions on Information Theory*. Vol. 22. 6. 1976, pp. 644–654. DOI: 10.1109/TIT.1976.1055638.

[6] Clarence Filsfils et al. "The Segment Routing Architecture". In: *Proceedings of the IEEE Global Communications Conference (GLOBECOM)*. 2015, pp. 1–6. DOI: 10.1109/GLOCOM.2015.7417368.

[7] Lixin Gao. "On Inferring Autonomous System Relationships in the Internet". In: *IEEE/ACM Transactions on Networking* 9.6 (2001), pp. 733–745.

[8] P. Brighten Godfrey et al. "Pathlet Routing". In: *Proceedings of the ACM SIGCOMM Conference*. Vol. 39. 4. 2009, pp. 111–122. DOI: 10.1145/1592568.1592596.

[9] Fabian Hauser et al. "P4-IPsec: Site-to-Site and Host-to-Site VPN with IPsec in P4-Based SDN". In: *IEEE Access* 8 (2020), pp. 139567–139586.

[10] Fabian Hauser et al. "P4-MACsec: Dynamic Topology Monitoring and Data Layer Protection with MACsec in P4-Based SDN". In: *IEEE Access* 8 (2020), pp. 58845–58858.

[11] Charles L. Hedrick. *Routing Information Protocol*. RFC 1058. 1988. URL: https://datatracker.ietf.org/doc/html/rfc1058.

[12] Cristian Hesselman et al. "A Responsible Internet to Increase Trust in the Digital World". In: *Journal of Network and Systems Management* 28.4 (2020), pp. 882–922. DOI: 10.1007/s10922-020-09564-7.

[13] Seok Hee Jeon and Sang Keun Gil. "Optical Secret Key Sharing Method Based on Diffie–Hellman Key Exchange Algorithm". In: *Journal of the Optical Society of Korea* 18.5 (2014), pp. 477–484. DOI: 10.3807/JOSK.2014.18.5.477.

[14] Morteza Kheirkhah et al. "UCIP: User Controlled Internet Protocol". In: *Proceedings of IEEE INFOCOM 2020 Workshops*. 2020, pp. 279–284. DOI: 10.1109/INFOCOMWKSHPS50562.2020.9162833.

[15] Changhoon Kim et al. "In-band Network Telemetry via Programmable Dataplanes". In: *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (SOSR)*. 2015, pp. 85–90. DOI: 10.1145/2751319.2751328.

[16] Tiffany Hyun-Jin Kim et al. "Lightweight Source Authentication and Path Validation". In: *Proceedings of the ACM SIGCOMM Conference*. 2014, pp. 271–282. DOI: 10.1145/2619239.2626323.

[17] Vasileios Kotronis, Christos Xenofontas Dimitropoulos, and Bernhard Ager. "Outsourcing the Routing Control Logic: Better Internet Routing Based on SDN Principles". In: *Proceedings of the 11th ACM Workshop on Hot Topics in Networks (HotNets-XI)*. 2012, pp. 55–60. DOI: 10.1145/2390231.2390241.

[18] Yichen Li et al. "InvisiFlow: Practical Source-Controlled Forwarding with Path Privacy". In: *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2025. URL: https://www.usenix.org/conference/nsdi25/presentation/li-yichen.

[19] Rami Lychev, Sharon Goldberg, and Michael Schapira. "BGP Security in Partial Deployment: Is the Juice Worth the Squeeze?" In: *ACM SIGCOMM Computer Communication Review* 43.4 (2013), pp. 171–182. DOI: 10.1145/2486001.2486002.

[20] Ratul Mahajan, David Wetherall, and Thomas E. Anderson. "Mutually Controlled Routing with Independent ISPs". In: *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2007, pp. 355–368.

[21] John Moy. *OSPF Version 2*. RFC 2328. 1998. URL: https://datatracker.ietf.org/doc/html/rfc2328.

[22] Jad Naous et al. "Verifying and Enforcing Network Paths with ICING". In: *Proceedings of the 7th ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*. 2011, p. 30. DOI: 10.1145/2079296.2079326.

[23] Isaac Oliveira et al. "dh-aes-p4: On-Premise Encryption and In-Band Key-Exchange in P4 Fully Programmable Data Planes". In: *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2021, pp. 148–153. DOI: 10.1109/NFV-SDN53031.2021.9665012.

[24] Konstantinos Papadopoulos, Panagiotis Papadimitriou, and Chrysa Papagianni. "PFA-INT: Lightweight In-Band Network Telemetry with Per-Flow Aggregation". In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2021, pp. 60–66. DOI: 10.1109/NFV-SDN53031.2021.9665016.

[25] Adrian Perrig et al. *SCION: A Secure Internet Architecture*. Springer International Publishing, 2017. ISBN: 978-3-319-67080-5.

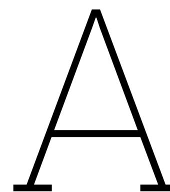
[26] Yakov Rekhter, Tony Li, and Susan Hares. "A Border Gateway Protocol 4 (BGP-4)". In: *RFC 4271* (2006). URL: https://datatracker.ietf.org/doc/html/rfc4271.

[27] Yakov Rekhter, Tony Li, and Susan Hares. *A Border Gateway Protocol 4 (BGP-4)*. RFC 4271. 2006. URL: https://datatracker.ietf.org/doc/html/rfc4271.

[28] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. RFC Editor, Aug. 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/rfc/rfc8446.

[29] Vytautas Valancius et al. "Wide-Area Route Control for Distributed Services". In: *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*. 2010, pp. 17–30.

[30] Daniel Wong, Aleksandrs Slivkins, and Emin Gün Sirer. "Truth in Advertising: Lightweight Verification of Route Integrity". In: *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. 2007.



P4 Header and Digest Definitions

This appendix lists the key P4 struct and header definitions from `hermes_line.p4` for reference.

A.1. Header Types

Listing A.1: Base and data packet headers in `hermes_line.p4`.

```
header hermes_base_t {
    bit<8>  msg_type;     // 0=DATA, 1=VERIFY, 2=RESPONSE
    bit<8>  version;      // protocol version = 1
    bit<16> length;       // total packet length
}

header hermes_data_t {
    bit<32> accumulator;     // updated at each hop
    bit<16> hop_count;       // hops completed (= hop index at this switch)
    bit<32> flow_id;         // identifies path/session
    bit<32> seq;             // monotonically increasing sequence number
    bit<48> timestamp_ms;    // sender timestamp in milliseconds
    bit<32> nonce;           // per-flow random nonce
}
```

A.2. Digest Type

Listing A.2: Digest emitted at the final hop.

```
struct hermes_digest_t {
    bit<32> flow_id;
    bit<16> hop_count;
    bit<32> accumulator;
    bit<32> seq;
    bit<48> timestamp_ms;
    bit<32> nonce;
    bit<1>  trigger_dh;    // 1 = key pool exhausted, renegotiate
    bit<32> hop_idx;       // switch that triggered renegotiation (0-2)
}
```

A.3. Opcode Definitions

Listing A.3: 3-bit opcode constants.

```
const bit<8> OP_ADD    = 0;  // acc + key
const bit<8> OP_ROT_AB = 2;  // rotl(acc, key)
```

```
const bit<8> OP_AND    = 3;  // acc & key
const bit<8> OP_OR     = 4;  // acc | key
const bit<8> OP_SUB_BA = 5;  // key - acc
const bit<8> OP_SUB_AB = 6;  // acc - key
const bit<8> OP_XOR    = 7;  // acc ^ key
// OP_ROT_BA (1): rotl(key, acc) -- else branch in do_accumulate
```

B

Hermes Server Wire Protocol

All communication between the controller and the Hermes server uses newline-delimited UTF-8 application messages over an mTLS-protected TCP connection on port 5555.

B.1. DH Initialisation (DH_INIT / DH_RESP)

Listing B.1: DH initialisation request and response.

```
-- Request (controller -> server):
DH_INIT <sw_id> <G> <P> <N> <Ka_0> <Ka_1> ... <Ka_{N-1}>

-- Response (server -> controller):
DH_RESP <N> <Kb_0> ... <Kb_{N-1}> <op_0> ... <op_{N-1}> <lfsr_seed>
```

Where:

- `sw_id`: switch name (e.g. `s1`, `s2`, `s3`)
- `G`, `P`: 32-bit public generators (decimal)
- `N`: number of keys, $N \in [330, 350]$
- `Ka_i`: switch public key i: $(G \oplus P)$ & B_i
- `Kb_i`: server public key i: $(G \oplus P)$ & A_i
- `op_i`: 3-bit opcode for key i (decimal 0–7)
- `lfsr_seed`: 32-bit non-zero LFSR seed (decimal)

B.2. Key Regeneration (DH_KEY / DH_KEY_RESP)

Identical in format to `DH_INIT` / `DH_RESP` with command names changed:

Listing B.2: Key regeneration request and response.

```
DH_KEY <sw_id> <G> <P> <N> <Ka_0> ... <Ka_{N-1}>
DH_KEY_RESP <N> <Kb_0> ... <Kb_{N-1}> <op_0> ... <op_{N-1}> <lfsr_seed>
```

B.3. Verification (VERIFY / RESULT)

Listing B.3: Verification request and response.

```
-- Request (controller -> server):
VERIFY <flow_id> <hop_count> <acc_final> <seq> <timestamp_ms> <nonce>

-- Response (server -> controller):
```

```
RESULT ACCEPT
RESULT REJECT
```

The server rejects the request (returning `RESULT REJECT`) if:

- `timestamp_ms` is more than 30 000 ms from the server's current time;
- `(flow_id, seq)` has previously been accepted; or
- the independently computed accumulator does not equal `acc_final`.