Ticket to Ride: Locally Steered Source Routing for the Lightning Network

Sajjad Alizadeh
Department of Electrical and Computer Engineering, University of Alberta, Canada

Majid Khabbazian
Department of Electrical and Computer Engineering, University of Alberta, Canada

Abstract

Route discovery in the Lightning Network is challenging because senders observe only static channel capacities while real-time balances remain hidden. Existing locally steered schemes such as SpeedyMurmurs protect path privacy but depend on global landmark trees whose maintenance traffic and detours inflate latency and overhead.

We present *Ticket to Ride* (T2R), a locally steered source-routing framework that encodes the set of channels a payment *may* traverse into a compact *ticket*—an approximate-membership filter keyed with per-hop Diffie–Hellman secrets. Each relay learns only whether its *own* outgoing edges are permitted, yielding the same incident-edge privacy as SpeedyMurmurs while eliminating the need to build and maintain global landmark trees or any other shared routing state.

Extensive simulations on real snapshots—incorporating churn, silent shutdowns, and random channel saturation—show that T2R boosts end-to-end success by up to 9% and cuts median delay by 1.6× relative to SpeedyMurmurs, all with <1 kB total overhead and *no* extra handshakes. Because tickets are processed hop-by-hop and can be prefixed by a trampoline, T2R remains lightweight enough for resource-constrained IoT nodes.

2012 ACM Subject Classification Networks

Keywords and phrases Lightning Network, Source Routing, Approximate Membership Filters

Digital Object Identifier 10.4230/LIPIcs.AFT.2025.27

1 Introduction

Payment channel networks (PCNs) such as the Lightning Network (LN) [14] scale blockchains by locking funds into two-party channels and allowing unlimited off-chain transfers that settle on-chain only when channels are opened or closed. By moving most activity off the base ledger, PCNs greatly increase throughput and lower fees, making cryptocurrency payments competitive with traditional systems.

When a payer lacks a direct channel to the payee, the payment must be forwarded through intermediaries. The public LN gossip protocol reveals the topology and the fixed *capacity* of every channel, yet keeps each side's *balance* private. Consequently, path finding is hard: a route that appears viable may fail at run time because an intermediate hop has insufficient outgoing liquidity.

All major LN implementations currently employ *source routing*, including LND [11], Core Lightning [3], and Eclair [1]. The sender uses a shortest-path search on the public graph, packs the resulting hop list into an onion (BOLT #4, the LN's onion routing specification) [10], and launches the payment. This approach leaves intermediaries passive and often requires repeated probing or guessing, so routing remains a bottleneck.

The literature now recognizes three broad classes of routing algorithms: (i) algorithms that keep all control at the source, (ii) multi-path "streaming" schemes that split a payment across several precomputed paths, and (iii) *local algorithms* in which nodes along the path can steer the payment by choosing the next hop on the fly.

Prominent local algorithms include SilentWhispers [12] and SpeedyMurmurs [17]. In each, intermediaries consult local information to influence the forwarding direction, while the destination node has been kept concealed from them. Both SilentWhispers and SpeedyMurmurs share a common architectural premise: they construct an auxiliary overlay—rooted landmark-trees in SilentWhispers and VOUTE-style spanning trees in SpeedyMurmurs. More specifically, SpeedyMurmurs assigns every node a compact *coordinate* vector. Greedy forwarding then reduces to choosing a neighbor that monotonically improves this coordinate with respect to the destination. While SilentWhispers and SpeedyMurmurs are conceptually similar, SpeedyMurmurs refines the idea along several axes: it eliminates the heavy multi-party distance-vector computation of SilentWhispers, shrinks routing state, and achieves lower communication latency in micro-benchmarks and real-world snapshots [17]. Following the authors' empirical evidence that their embedding outperforms prior landmark routing both in success rate and end-to-end delay, we adopt SpeedyMurmurs as the representative local-steering baseline in this work.

Although SpeedyMurmurs achieves low latency and strong path privacy, it imposes a control-plane burden: multi-tree bootstrap floods, periodic stabilisation beacons, and per-tree state at every relay. Each drain or failure of a parent edge forces cascaded coordinate updates, and nodes adjacent to a landmark become fee and censorship choke points. Greedy forwarding on the embedding can also detour from shortest paths. These inefficiencies motivate a lighter local-steering scheme that carries *all* routing hints within each payment itself, removing global trees entirely.

In this paper, we introduce Ticket to Ride (T2R), a locally steered source routing framework that eliminates the need for global landmarks or coordinate trees. We implement this idea with a compact, cryptographically keyed *ticket*: a privacy-preserving set-membership structure that lists exactly which outgoing channels the sender allows. Concretely, the ticket is instantiated using an approximate-membership query (AMQ) filter—such as a Bloom filter or Golomb–Rice coded set—that trades a tunable false-positive rate for sub-kilobyte size. Upon receipt, a relay derives a one-hop secret from the sender's ephemeral public key, hashes each of its outgoing channels under that secret, and tests ticket membership. All channels that match are authorized; the relay then ranks them using a local policy—e.g., lowest cumulative fee if the sender flags cost sensitivity, or highest success probability—and forwards on the top candidate. Links invisible to the filter are ignored, and the keying prevents a relay from inferring which edges are authorized for other nodes. The ticket thus reveals neither the destination's identity nor the remaining route, guarantees monotonic progress toward the receiver, and eliminates the churn, choke points, and detour risk inherent in landmark-tree routing.

Forwarding depends only on per-hop membership tests, so a ticket remains useful even if its channel list is not up to date[1]: a vanished edge is silently skipped, and an unseen new edge merely foregoes a shortcut. In any case, channel openings and closures are already advertised through the routine gossip traffic that every node receives, so the information needed to refresh the filter circulates at no extra network cost. Because the ticket lists only admissible edges, the sender can blacklist nodes or boost preferred channels—a level of control landmark trees cannot offer. The ticket piggybacks on the handshake message that neighboring relays already exchange, so it introduces no extra latency or bandwidth. If a

[1] Empirically, Lightning's topology evolves relatively slowly—channel openings and closures typically number in the hundreds per day [8], thus a snapshot taken approximately one day earlier usually differs from the current graph by roughly 1% of edges.

probe fails, the sender drops the offending node and rebuilds the ticket; the new ticket omits that identifier and guarantees the payment will never be forwarded there again. Such explicit, sender-driven blacklisting is *infeasible* in landmark-tree and other purely local algorithms, where each intermediate node selects its next hop autonomously and the sender has no way to forbid a particular relay.

SpeedyMurmurs is appealing for lightweight nodes, as it shifts mapping responsibilities away from the sender, allowing them to remain passive. However, our ticket-based approach can also accommodate resource-constrained endpoints—such as IoT sensors or mobile wallets that frequently remain offline. Specifically, we propose a lightweight variant called *segment tickets with trampolines*, exploiting the draft trampoline extension [5]. Here, the sender delegates topology management by selecting a short chain of always-online trampoline nodes, each responsible for crafting fresh tickets for their respective segments. Thus, trampolines shoulder the workload of maintaining updated channel filters, making this mode particularly suitable when *both* sender and receiver have limited memory or infrequent connectivity to process topology updates themselves.

As a final contribution, we validate the design through extensive, trace-driven simulations on recent LN snapshots. Across tens of thousands of random payments, T2R achieves consistently higher success probability and lower end-to-end delay than SpeedyMurmurs, with the margin widening in the most demanding scenarios: long source–destination pairs, heavily saturated channels, and bursty link churn. These results confirm that removing landmark trees not only lightens the control plane but also yields tangible data-plane gains in practical, large-scale topologies such as LN.

2 System Model

Overlay graph. We represent the payment-channel network as an undirected multigraph $G = (V, E)$. Each vertex is a node identified by a long-term public key, and each edge $e = (u, v) \in E$ denotes a bidirectional channel of fixed capacity $C_{uv} \in \mathbb{N}$. The directional balances $b(u \to v)$ and $b(v \to u)$ are private, dynamic values that satisfy $b(u \to v) + b(v \to u) = C_{uv}$.

Gossip and topology freshness. Channel openings, closures, and capacity changes are disseminated by the ordinary LN gossip protocol and reach every honest node within Δ_{gossip}. Our protocol reuses this information and introduces *no additional control-plane traffic*, contrasting with the tree-stabilisation beacons required by SpeedyMurmurs.

Communication model. Nodes exchange authenticated, FIFO messages. Delivery latency is bounded by an unknown constant Δ; liveness uses this bound, while security arguments remain delay-agnostic.

Adversary. A static Byzantine adversary corrupts any subset $V_{\mathcal{A}} \subset V$ with $|V_{\mathcal{A}}| \leq f|V|$ for constant $f < 1$, mirroring the threat model of SpeedyMurmurs. Corrupted nodes may drop, delay, or modify messages and deplete balances but cannot break standard cryptography.

Payments and tickets. To transfer an amount x from a source s to a destination d, the sender attaches a cryptographically keyed *ticket* τ to the payment packet. The ticket τ is a privacy-preserving approximate-membership filter that encodes precisely the set $\mathcal{L}_{s,d} \subseteq E$ of edges the sender authorizes for forwarding.

Local forwarding rule. When a relay u receives $\langle x, \tau \rangle$, it derives a one-hop secret s_u from the sender's ephemeral public key, hashes each adjacent edge (u, v) under that secret, and tests membership in τ. Edges that match are ranked by a local policy (e.g. lowest cumulative fee or highest empirical success rate); the highest-ranked authorized edge is chosen as the next hop. No parent pointers or coordinate updates are maintained.

3 T2R Routing

In T2R , the sender appends to each payment a compact probabilistic set-membership filter—called a *ticket*—that, under a per-payment secret key, encodes the set $\mathcal{L}_{s,d}$ of channels the sender authorizes. The ticket is piggy-backed on the first control frame exchanged by neighboring nodes. Upon receipt, a relay hashes every outgoing channel with the secret, tests ticket membership, ranks the authorized matches according to its local policy (e.g., lowest fee or highest success probability), and forwards the payment on the top-ranked edge.

Despite its modest implementation footprint, the mechanism is expressive: it can emulate SpeedyMurmurs without constructing landmark trees. A node locally builds the trees, collects the union of edges that greedy forwarding might traverse, and inserts those identifiers into the ticket. The converse does not hold—landmark embeddings cannot realize sender-defined blacklists or per-edge priorities without redesigning the entire coordinate system.

In the remainder of this section, we first present a generic framework for constructing tickets, then formalize our privacy target—*incident-edge privacy*—and prove that any ticket generated by this framework satisfies it. We next introduce *TwinTicket*, a two-filter refinement that shrinks the filter size required to reach a given success rate.

3.1 Ticket Design

Our *ticket framework* comprises two algorithms: Issue, executed by an *originator* (typically the sender), and Open, executed by each intermediate node. The framework relies on two standard primitives:

- A hash function $H : \{0,1\}^* \to \{0,1\}^\kappa$, modelled as a random oracle and used solely as a source of pseudorandom bits.
- An **approximate-membership query structure (AMQ)** that supports three polynomial time operations:
  - Create(m, α): initializes an empty filter sized for up to m insertions with false-positive probability at most α;
  - Insert(S, e): adds element e to filter S;
  - Query(S, e): returns `present` with probability $\leq \alpha$ if $e \notin \mathsf{S}$, and never returns `absent` for an inserted element.

For each node u controlling an authorized channel, the originator derives a unique per-hop secret using elliptic-curve Diffie–Hellman (ECDH). Specifically, the originator selects an ephemeral public key $\mathrm{PK}_{\mathsf{ep}}$, while each node u has a long-term public key PK_u published through the network gossip. The shared per-hop secret is computed as:

$$s_u \leftarrow \mathrm{DH}(\mathrm{PK}_{\mathsf{ep}}, \mathrm{PK}_u),$$

where DH denotes an ECDH key exchange, followed by a standard key derivation function (e.g., HKDF) to yield a uniform κ-bit key. Because the sender knows the ephemeral secret and node u holds the private key corresponding to PK_u, only these two parties can compute the shared secret s_u. Consequently, any ticket entries derived from s_u appear indistinguishable from random to all other nodes. This per-hop shared-secret derivation mirrors the mechanism already employed by LN's onion-routing protocol [10].

Issue Algorithm.

Algorithm 1 describes the ticket issuance process. The algorithm accepts as input:

(i) a set of authorized channels $\mathcal{C} \subseteq E$,
(ii) the public keys $\{\mathrm{PK}_u\}$ of nodes incident to these channels,
(iii) an ephemeral public key $\mathrm{PK}_{\mathsf{ep}}$ chosen by the sender,
(iv) a target false-positive probability ϵ.

The algorithm initializes an AMQ structure sized according to $|\mathcal{C}|$ and ϵ. For each channel $(u, v) \in \mathcal{C}$, the algorithm computes the per-hop secret s_u and inserts the hash value $H(v \parallel s_u)$ into the AMQ. The resulting AMQ structure is output as the ticket τ.

Algorithm 1 $\textsc{Issue}(\mathcal{C}, \{\mathrm{PK}_u\}, \mathrm{PK}_{\mathsf{ep}}, \epsilon)$

```
S ← AMQ.Create(|C|, ε)
for all (u, v) ∈ C do
  s_u ← DH(PK_ep, PK_u)
  AMQ.Insert(S, H(v ∥ s_u))
end for
return S // ticket τ
```

Open Algorithm.

Upon receiving a payment, an intermediate node executes Open to determine whether any of its outgoing channels are authorized by the ticket τ and, if so, to select one for forwarding. The algorithm takes as input:

(i) the ticket τ (AMQ structure) carried with the payment,
(ii) the node's own long-term public/secret key pair $(\mathrm{PK}_u, \mathrm{SK}u)$,
(iii) the sender's ephemeral public key PKep (included in the packet),
(iv) a ranking policy $\textsc{Rank}(\cdot)$ (e.g., lowest fee or highest empirical success rate).

First, the node derives its per-hop secret $s_u \leftarrow \mathrm{DH}(\mathrm{PK}_{\mathsf{ep}}, \mathrm{PK}_u)$. For each outgoing channel $(u, w) \in \mathrm{Adj}(u)$ it queries the ticket using the tag $H(w \parallel s_u)$. The set of channels for which the query returns `present` constitutes the authorized subset $\mathcal{A}_u$. If $\mathcal{A}_u$ is empty the node aborts the payment locally; otherwise it ranks $\mathcal{A}_u$ with Rank and forwards the payment along the highest-ranked edge.

Algorithm 2 $\textsc{Open}(\tau, \mathrm{PK}_{\mathsf{ep}}, \mathrm{PK}_u, \mathrm{SK}u)$

```
s_u ← DH(PKep, PK_u)
A_u ← ∅
for all (u, w) ∈ Adj(u) do
  y ← H(w ∥ s_u)
  if AMQ.Query(τ, y) == present then
    A_u ← A_u ∪ (u, w)
  end if
end for
if A_u = ∅ then
  abort // no authorized edge
else
  (u, w*) ← Rank(A_u)
  forward payment over (u, w*)
end if
```

The choice of RANK is intentionally left flexible. At a minimum, any channel (u, w) whose local balance cannot cover the payment amount is excluded from consideration. Beyond that filter, RANK can implement a variety of policies: it may prefer the lowest-fee edge, adopt the heuristics proposed in Flare [16], or emulate Spider-style congestion metrics [19]. Nevertheless, because each node independently selects its forwarding edge based solely on local information, the resulting route need not be globally optimal. Designing an effective ranking strategy is orthogonal to the ticket mechanism itself and remains open for future exploration. In our evaluation, we simply select uniformly at random among the authorized edges with sufficient balance.

Incident-edge privacy.

A well-formed ticket must limit what any relay can infer about the remainder of the route. Intuitively, an intermediate node u should learn only whether *its own* channels are authorized and nothing about edges it does not control, beyond the unavoidable disclosure of the ticket's overall size. We capture this requirement with the indistinguishability notion below. Let τ denote the real ticket for a payment (s, d, x) and let $\tilde{\tau}_u$ be a *semi-ticket* that keeps u's authorized edges intact while masking every other authorized edge by an independent random λ-bit string, so that τ and $\tilde{\tau}_u$ contain the same number of inserts. A ticket scheme satisfies *incident-edge privacy* if no probabilistic polynomial-time adversary controlling u can distinguish τ from $\tilde{\tau}_u$ with more than negligible advantage in the security parameter λ.

▶ **Definition 1** (Incident-edge privacy). *Let τ be the ticket issued for a payment (s, d, x) and fix an intermediate node $u \in V$. Construct a* semi-ticket $\tilde{\tau}_u$ *in two steps:*

(1) Insert into an empty AMQ every authorized channel incident to u.
(2) Insert uniformly random κ-bit strings in place of all other authorized channels, so that τ and $\tilde{\tau}_u$ contain the same number of inserts.

A ticket scheme satisfies incident-edge privacy *if, for every probabilistic polynomial-time adversary A controlling only node u,*

$$\left|\Pr[A(\tau) = 1] - \Pr[A(\tilde{\tau}_u) = 1]\right| \;\leq\; \mathrm{negl}(\kappa),$$

where the probabilities are over the randomness of ticket generation and of A, and $\mathrm{negl}(\kappa)$ *is negligible in the security parameter κ.*

The definition formalizes the guarantee that, apart from its own adjacent channels and the ticket's length, an intermediate node gains no information about which other edges the sender has authorized.

▶ **Theorem 2** (Incident-edge privacy of the ticket framework). *Assume the hash function H is modeled as a random oracle and the Diffie–Hellman key exchange is a secure key-agreement protocol. Then every ticket generated by the* ISSUE *algorithm in our framework satisfies the incident-edge privacy property of Definition 1.*

Proof sketch. Fix an intermediate node $u \in V$ and a probabilistic polynomial-time distinguisher A that corrupts u. Let κ denote the global security parameter, and let $q_H(\kappa)$ bound the number of random-oracle queries that A may issue.

Game G_0 (real world). The challenger runs ISSUE to generate the genuine ticket τ and hands $(\tau, \mathrm{PK}_{\mathsf{ep}})$ to A, which can query the oracle before outputting a bit.

Game G_1 (ideal world). As in G_0, except that for every authorized channel $(u', v') \notin \mathrm{Adj}(u)$ the challenger replaces the tag $H(v' \| s_{u'})$ in τ with an independently uniform κ-bit string. The resulting structure is exactly the semi-ticket $\tilde{\tau}_u$ of Definition 1.

Indistinguishability of G_0 and G_1. The two games differ only in tags for non-adjacent channels. For such an edge, the real tag in G_0 is

$$T = H\big(v' \| s_{u'}\big), \quad s_{u'} = \mathrm{DH}(\mathrm{PK}_{\mathsf{ep}}, \mathrm{PK}_{u'}).$$

Because A lacks the secret key of $\mathrm{PK}_{u'}$, the shared secret $s_{u'}$ is computationally indistinguishable from a uniform κ-bit string. In the random-oracle model, unless A queries H on the exact pre-image $v' \| s_{u'}$, the value T is random. Replacing T with an independent random string, as in G_1, therefore changes A's view only if it guesses $v' \| s_{u'}$.

The probability that A hits any specific κ-bit suffix within at most $q_H(\kappa)$ oracle queries is $q_H(\kappa)\, 2^{-\kappa} = \mathrm{negl}(\kappa)$. A union bound over all non-adjacent channels preserves negligibility, so

$$\big|\Pr[A^{G_0}(1) = 1] - \Pr[A^{G_1}(1) = 1]\big| \le q_H(\kappa)\, 2^{-\kappa} = \mathrm{negl}(\kappa).$$

Because the adversary's advantage is negligible, every ticket produced by Issue satisfies Definition 1. ◀

3.2 TwinTicket

The basic ticket of Section 3.1 stores all authorized channels in a *single* AMQ. Improving its accuracy normally means lowering the false-positive rate, which in turn expands the filter. TwinTicket takes an alternative path: it embeds two AMQs—one that *includes* authorized channels and a second that explicitly *excludes* those few channels misclassified by the first. In other words, TwinTicket contains

- an **inclusion filter** $\mathsf{S}^{\mathsf{inc}}$, which holds every authorized edge $(u, v) \in \mathcal{L}_{s,d}$; and
- an **exclusion filter** $\mathsf{S}^{\mathsf{exc}}$, which holds each false positive discovered when the sender queries $\mathsf{S}^{\mathsf{inc}}$ locally. A channel (u, v) is inserted in encoded form $H\big(v \| s_u\big)$, identical to the inclusion encoding.

Forwarding rule. A relay u first tests every adjacent edge against $\mathsf{S}^{\mathsf{inc}}$. If the result is `present`, the relay performs a second test against $\mathsf{S}^{\mathsf{exc}}$; the edge is deemed authorized only if it is *not* found there. The relay then ranks all authorized edges using its local policy and forwards on the highest-ranked candidate.

Benefits. Both the single-filter ticket and the double-filter TwinTicket grow in size as the target false positive reduces. Simulations in Section 5 show that TwinTicket achieves the same success probability and delay as a finely tuned single AMQ while using fewer bits. *This space saving does not weaken privacy.* Because every unauthorized edge is still added—again in hashed form—only by its own endpoint, TwinTicket preserves incident-edge privacy: an intermediate node u learns solely whether each of *its* channels is allowed or blocked, now with higher classification accuracy.

4 Overhead

Unlike SpeedyMurmurs, T2R dispenses with landmark trees altogether, yet it can reproduce SpeedyMurmurs' forwarding behavior—simply insert into the ticket every edge the payment is authorized to traverse. The reverse is impossible: landmark trees cannot (i) exclude specific

channels or nodes, (ii) delegate route selection to third parties (e.g. the receiver), or (iii) guarantee shortest-path delivery without rebuilding the tree.

The price we pay is the *ticket*: it must be (i) generated by an originator and (ii) carried once with the payment. This section quantifies the computational, and communication overhead of those two steps. We start with analyzing the ticket size.

4.1 Ticket Size

The size of a ticket—i.e., the bit length of the AMQ structure that travels with each payment—depends on three parameters:

(i) the particular *AMQ design* employed;
(ii) the target false-positive bound α;
(iii) the number of elements inserted, m.

The literature offers a rich family of AMQ designs, each with its own space–functionality trade-off. Classic examples include the Bloom filter [4]; the counting Bloom filter, which supports deletions [7]; the Cuckoo filter [6]; and the Quotient filter [2]. Golomb–Rice coded sets (GCS)—used in Bitcoin's compact block filters—achieve near-optimal compression for static sets by entropy-coding sorted hashes [15]. More recent proposals, such as the Learned Bloom filter [9], trade a small probability of false negatives for additional space savings. For fixed set size m and target false-positive rate α, structures like the Cuckoo filter, quotient filter, or GCS can be more compact than the original Bloom filter but may incur higher insertion cost, require sorted input, or demand more intricate parameter tuning.

For example, the classic *Bloom filter*—introduced by Bloom in 1970—supports only two operations, Insert and Query, and provides no deletion primitive. With m insertions and target false-positive probability α, its information-theoretically optimal length is

$$\ell_{\text{Bloom}} \;=\; \left\lceil -\frac{m\,\ln\alpha}{(\ln 2)^2} \right\rceil \;\approx\; 1.44\,m\,\log_2\!\big(1/\alpha\big)\ \text{bits}, \tag{1}$$

obtained with $k = \lceil (\ell_{\text{Bloom}}/m)\,\ln 2 \rceil$ independent hash functions [4].

A second widely deployed AMQ is the *Golomb–Rice Coded Set* used in Bitcoin block filters (BIP 158) [15]. A GCS stores the sorted hash values and compresses their gaps with Golomb–Rice coding, again supporting Insert and Query but requiring no random oracle. Choosing the Golomb–Rice parameter $P = \lceil \log_2(1/\alpha) \rceil$ yields a false-positive rate at most α and an expected length

$$\ell_{\text{GCS}} \;=\; m\,(P+1) \;=\; m\big(\lceil \log_2(1/\alpha) \rceil + 1\big)\ \text{bits}. \tag{2}$$

For the practical false–positive window $\alpha \in [2^{-7},\, 2^{-15}]$ ticket overhead remains modest. In a GCS the expected length per element is $P+1$ bits, where $P = \lceil \log_2(1/\alpha) \rceil$. Thus each authorized channel costs 8 bits when $\alpha = 2^{-7}$ and 16 bits when $\alpha = 2^{-15}$—only 1–2 bytes apiece. By contrast, a Bloom filter needs $\ell_{\text{Bloom}} \approx 1.44 \log_2(1/\alpha)$ bits per element: about 10 bits (≈ 1.3 B) at $\alpha = 2^{-7}$ and 22 bits (≈ 2.7 B) at $\alpha = 2^{-15}$. The dominant contributor to ticket length is therefore the element count m: in the practical range $\alpha \in [2^{-15}, 2^{-7}]$ each additional authorized channel adds between $c{=}1$ and $c{=}2$ bytes.

If the ticket contains only the channels of a *single* source-to-destination path, our framework reduces to the single-route source routing that all major Lightning implementations already employ. Adding more channels, however, gives intermediate nodes additional "steering" options and thus lowers the probability that an in-flight payment stalls on an exhausted link.

A natural choice is to populate the ticket with the union of edges that lie on *any* shortest path between the sender and the receiver. Although many shortest paths can coexist, the ticket remains compact: on a recent public Lightning snapshot, authorizing about 126 channels suffices for 90% of randomly chosen source–destination pairs, and 611 channels cover 99% of such pairs.

4.2 Ticket Generation

Empirical measurements show that the public Lightning graph is remarkably stable: the average channel lifetime is nearly two hundred of days [21]. Because these modest topology updates propagate through the standard gossip protocol, a well-provisioned node can maintain an up-to-date view at virtually no extra cost. Building a ticket therefore needs only this topology snapshot; rapidly fluctuating balances are irrelevant.

Ultra-constrained senders (IoT tags, single-board wallets) cannot hold the full graph. We therefore combine our ticket mechanism with *trampoline payments* and slice the route into three *segment tickets*:

(a) $\tau_{S\to T_1}$ **(sender-issued)** covers the first few hops from the sender S to the entry trampoline T_1 and is keyed with an ephemeral $\mathrm{PK}_{\mathsf{ep}}^{(1)}$ chosen by S.

(b) $\tau_{T_1\to T_k}$ **(macro ticket, sender-issued)** encodes the ordered list of trampoline IDs $T_2,\ldots,T_k$, where T_k is supplied by the receiver (see below). Each trampoline T_i reads the next ID, computes its own micro-path to T_{i+1}, and attaches a fresh sub-ticket.

(c) $\tau_{T_k\to R}$ **(receiver-issued)** steers the last one or two hops from T_k to the receiver R, optionally biasing the route toward R's best-funded inbound channel. It is keyed with a second ephemeral $\mathrm{PK}_{\mathsf{ep}}^{(2)}$ chosen by R.

This approach brings the following advantages:

- **Compactness.** The sender and receiver tickets span only a small number of hops (typically one or two) and the macro ticket is merely two or three compressed pubkeys (< 70 B). Total header overhead, therefore, stays well under one kilobyte.
- **Balanced computation.** Lightweight senders route only to T_1; each trampoline solves a local micro-routing instance; only trampolines need the global graph.
- **Privacy.** Every ticket is keyed with a fresh Diffie–Hellman secret for each hop, so an intermediate node learns only whether *its* outgoing edges are permitted. All other tags look uniformly random, achieving the same incident-edge privacy as the original T2R.
- **Fine-grained channel control.** When constructing the final segment of the ticket, the receiver can privilege high-liquidity inbound channels or blacklist unreliable ones. Because this choice is encoded locally in the ticket and masked by the per-hop key, neither the sender nor any trampoline node learns which channels were favored.
- **Minimal invoice overhead.** When R responds to an `invoice_request` (BOLT #12) or encodes a BOLT #11 invoice, it appends the triple $\left(T_k, \mathrm{PK}_{\mathsf{ep}}^{(2)}, \tau_{T_k\to R}\right)$. The extra payload is ≤ 100 bytes for the ticket plus 33 bytes for the key—well below the 1 kB capacity of a level-10 QR code—so invoice usability is unchanged.

With this three-ticket trampoline design, even kilobyte-scale devices can therefore utilize locally steered, privacy-preserving routing without storing the complete Lightning topology.

4.3 Communication and latency overhead

When an intermediate node u forwards a payment over its channel to peer v, it engages in the standard Lightning commitment handshake. The procedure consists of two full-duplex rounds:

(1) u sends an `update_add_htlc` (carrying the onion payload) immediately followed by `commitment_signed`.
(2) After validating the update, v replies with `commitment_signed` and `revoke_and_ack`, completing the commit–reveal exchange.

The ticket is piggy-backed on the first `update_add_htlc`; no extra round trips are introduced, and only a few hundred bytes are appended to a frame that already averages ∼1500 bytes. Importantly, v need not wait for the entire ticket to arrive before it can respond: the Lightning spec allows a node to send its `commitment_signed` as soon as the new Hashed Timelock Contracts (HTLC) has been parsed and validated, while the remaining payload bytes stream in.

▶ **Example 3** (Latency impact)**.** Assume a single Lightning hop with a round-trip propagation delay of 80 ms. Because a commitment update consists of two such round trips, the propagation component alone contributes 160 ms to handshake latency.

Typical ticket ($m = 125,\ \alpha = 0.001\%$)**.** For a GCS filter with $\alpha = 0.00001$ the Golomb parameter is $P = \lceil \log_2(1/\alpha) \rceil = 17$, so the ticket length is

$$\ell = m(P + 1) = 125 \times 18 = 2\,250 \text{ bits} \approx 280 \text{ bytes}.$$

At 10 Mbit/s this transmits in $t = 2\,250/(10 \times 10^6) \approx 0.22$ ms, seven orders of magnitude smaller than one RTT.

Worst-case ticket ($m = 60\,000,\ \alpha = 0.001\%$)**.** If the sender naively inserted *all* public channels the ticket would be

$$\ell = 60\,000 \times 18 = 1\,080\,000 \text{ bits} \approx 130 \text{ kB},$$

which transmits in $t = 1\,080\,000/(10 \times 10^6) = 108$ ms, still below a single RTT.

We note that in LN the ticket travels piggy-backed on the initial `update_add_htlc`; subsequent frames (`commitment_signed`, `revoke_and_ack`) need not wait for the ticket to finish. SpeedyMurmurs defines latency as the duration of the *longest chain of causally dependent messages* in the protocol execution. Because the ticket is *outside* that critical chain, its transfer time—whether 0.5 ms or 48 ms—is absorbed during idle link time, leaving the overall handshake delay dominated by the 160 ms propagation budget.

4.4 Payment Splitting

SpeedyMurmurs can split a payment into partial payments, and send each partial payment along a different landmark–embedding tree, explicitly to *hide the total value from every individual relay* [17]. Achieving this goal requires the protocol to maintain several spanning trees and to send bookkeeping beacons whenever channels appear, disappear, or exhaust liquidity.

Our locally steered source routing realizes the same value privacy at virtually *no extra cost*. The sender simply issues as many tickets as there are partial payments—each keyed

with its own per–hop secret—and launches the sub-payments concurrently. No additional control traffic is needed, and ticket construction remains fully local.

In addition, because the sender controls each ticket's edge set, it can impose *explicit path separations*: for two partial payments, the sender may generate disjoint channel sets whenever the topology allows, thus guaranteeing that no single relay observes every share. SpeedyMurmurs cannot enforce such disjointness even when multiple landmark-embedding tree is used.

5 Simulation Methodology and Results

To quantify the practical impact of T2R, we built a LN simulator in Python[2] and ran all experiments on a dedicated 32-core workstation with 32, GiB RAM, Ubuntu 20.04 LTS, and Python 3.11. We benchmarked T2R against SpeedyMurmurs in its single-tree configuration, the variant that the original paper reports as offering the highest overall performance—namely, the greatest success rate and the lowest delay—when payments are not split across trees.

Input graph. All experiments use the public LN gossip dump exported by Pickhardt on 12 April 2022[3]. This snapshot coincides with the historical peak of publicly advertised LN adoption—both in nodes and in channels[4]—and has served as a benchmark in prior LN-routing work. [18] The raw graph contains **14 135** nodes, **60 757** bidirectional channels, and **9 087** unidirectional channels.

A Lightning *channel* is backed by a single 2-of-2 multisignature output of capacity C. At any moment this capacity is split between the two peers. Accordingly, our simulator models each bidirectional channel as two directed *half-channels*: $(u \to v)$ carries the balance held by u, and $(v \to u)$ carries the complementary balance owned by v (C minus u's share). Unidirectional channels are represented by a single directed edge.

Connectivity pruning. SpeedyMurmurs can embed its landmark tree only in a strongly connected graph, thus we restrict the snapshot to its largest strongly connected component, discarding the 1 002 nodes that lie in isolated islands. This pruning is required *only* for SpeedyMurmurs; T2R operates correctly on any topology. We also disable every channel the snapshot marks as inactive, ensuring that the simulator uses only links considered live.

Capacity and initial balances. The snapshot records the on-chain capacity C_{uv} (in satoshis) of each channel (u, v) but, as usual, not its private directional balances. For our *baseline* experiments we assume an even split and assign

$$b(u \to v) = b(v \to u) = \tfrac{1}{2} C_{uv}.$$

To study the effect of systematic skew we repeat the experiments with a fixed global ratio $p \in \{0.6, 0.7, 0.8, 0.9, 1.0\}$, setting

$$b(u \to v) = p\, C_{uv}, \qquad b(v \to u) = (1 - p)\, C_{uv}.$$

These five settings correspond to the (60:40) through (100:0) points on the x-axes of our figures.

Failure models. Lightning payments often fail because the sender's view is stale: a channel that *looks* usable in gossip may in fact be unusable when the HTLC arrives. We emulate two such hidden failure modes.

[2] `https://github.com/sajializ/ticket`
[3] `https://www.rene-pickhardt.de/listchannels20220412.json`
[4] `https://bitcoinvisuals.com/lightning`, chart "Total public channels / nodes (daily)"

Silent channel disablement. A fraction $\rho \in \{10, 20, 30, 40, 50\}\%$ of channels is randomly marked inactive—neither direction can forward—yet their gossip entries remain unchanged. This captures maintenance outages or force-closures that have not yet propagated.

Random saturation. We pick $k \in \{10, 20, 30, 40, 50\}\%$ of channels uniformly at random and set their forward balance to zero in one direction, $b(u \to v) = 0$, while the reverse edge still forwards. This models liquidity depletion rather than shutdown.

5.1 Payment Flow Generation

Our goal is to measure how effectively each algorithm discovers a route *when one truly exists* under the current liquidity snapshot. Accordingly, every data point in a batch of 50 000 payments is generated as follows.

1. **Endpoint sampling.** Pick source s and destination d uniformly at random from the pruned graph ($s \neq d$).
2. **Amount selection.** Draw a payment value of A according to the scenario:
   - One setting of our simulation picks a uniformly random amount between intervals $\{10^0\text{–}10^2, 10^2\text{–}10^3, 10^3\text{–}10^4, 10^4\text{–}10^5, 10^5\text{–}10^6\}$ satoshis for each payment.
   - All other settings pick a random amount A between 10^2–10^3 satoshis. We chose this interval for the amount of the payment because both algorithms have low differences regarding success rate using this interval, which enables us to examine how other behaviors of the network affect them.
3. **Feasibility of the payment.** Before routing, we verify that some *optimal* path can carry (s, d, A) under the current liquidity *and* each channel's HTLC policy:
   - Every announcement advertises `htlc_minimum_msat` and `htlc_maximum_msat`. A hop may forward only if `htlc_min` $\leq A \leq$ `htlc_max`.
   - A directed edge $(u \to v)$ must also have $\mathit{balance}(u \to v) \geq A$ at this moment.
   - Every channel on the optimal path must be active (not flagged as disabled).

   We run a shortest-path search that enforces *all* of the above constraints. The source and destination for which no feasible path exists are discarded and resampled, ensuring every payment is routable in principle, so any failure reflects routing performance, not an *impossible* payment.
4. **Algorithm trials.** Two independent copies of the network are made—one for T2R and one for SpeedyMurmurs. Each algorithm attempts to route (s, d, A) *once* on its private copy:
   a. If the attempt succeeds, the simulator decreases the directional balances along the path and increases the reverse directions, simulating an immediate settle.
   b. If it fails, no balances change.

   Because the two runs start from identical liquidity and evolve independently, their success probabilities can be compared without any interference.
5. **Ticket generation.** For every feasible request we enumerate *all* $s \to d$ paths that satisfy the public HTLC policy at every hop ($\texttt{htlc\_min}_{u\to v} \leq A \leq \texttt{htlc\_max}_{u\to v}$), and the advertised channel *capacity* is at least the payment amount A. Every channel–direction that appears on at least one such path is inserted into the ticket. To ensure that any residual failure arises from the algorithm rather than ticket collisions, we instantiate the filter with a near-zero false-positive rate. This setting makes the probability that an unusable edge is authorized negligible. We examine the false-positive rate on success ratio later in the simulation.

6. **Forwarding behavior** Once a feasible request (s, d, A) is admitted, each algorithm is invoked *exactly once*; a failure at any hop marks the entire payment as unsuccessful—no sender-side retry or multipath splitting is performed. Inside the network the two schemes behave differently:

   - **T2R.** Upon receiving the payment an intermediary node chooses one of its outgoing channels *uniformly at random*. If that channel both (i) meets the HTLC amount/balance constraints and (ii) appears in the ticket, the intermediary node forwards the HTLC along it. Otherwise the node samples another neighbor at random—without replacement—and repeats the test until either a suitable channel is found or all neighbors have been exhausted. If no channel passes the test the payment fails immediately.
   - **SpeedyMurmurs.** Each intermediary follows the greedy rule (as explained in their paper) of the landmark tree: forward to the unique neighbor whose tree distance to the destination is strictly smaller than its own.

This procedure isolates the algorithms' ability to *find* a viable route, not their capacity to judge when no route exists; it also keeps the liquidity landscape dynamic, as successful payments progressively update balances—reflecting real Lightning traffic.

5.2 Evaluation Metrics

From each batch of $n = 50\,000$ payment attempts we extract two quantities to compare both algorithms.

1. **Success probability.** For a batch of n independent payment attempts let $\mathcal{S} \subseteq \{1, \ldots, n\}$ be the index set of payments that are completed successfully. The empirical success rate is

$$P_{\text{success}} = \frac{|\mathcal{S}|}{n} = \frac{\#\text{successful payments}}{n}.$$

2. **Average hop count.** Denote by h_i the hop length of the *realized path* for a successful payment $i \in \mathcal{S}$. If $m = |\mathcal{S}|$ is the number of successes, the sample mean hop count is:

$$\bar{h} = \frac{1}{m} \sum_{i \in \mathcal{S}} h_i.$$

5.3 Results

Figures 1, 2 demonstrates the behavior of the T2R and the single-tree implementation of SpeedyMurmurs across four axes: payment amount (a), different initial balances (b), channel silent disablement (c), and random saturation (d).

Different payment amounts.

T2R sustains a $\geq 90\%$ first-attempt success rate up to 10^5 sat, whereas SpeedyMurmurs already drops to $\approx 77\%$ at 10^4 sat and collapses beyond 10^5 sat. Redundancy within the T2R 's multipath findings allows intermediate nodes to avoid unexpectedly empty links without sender involvement. The number of hops of T2R is considerably lower than SpeedyMurmurs, meaning for each payment size interval, the T2R finds shorter paths.

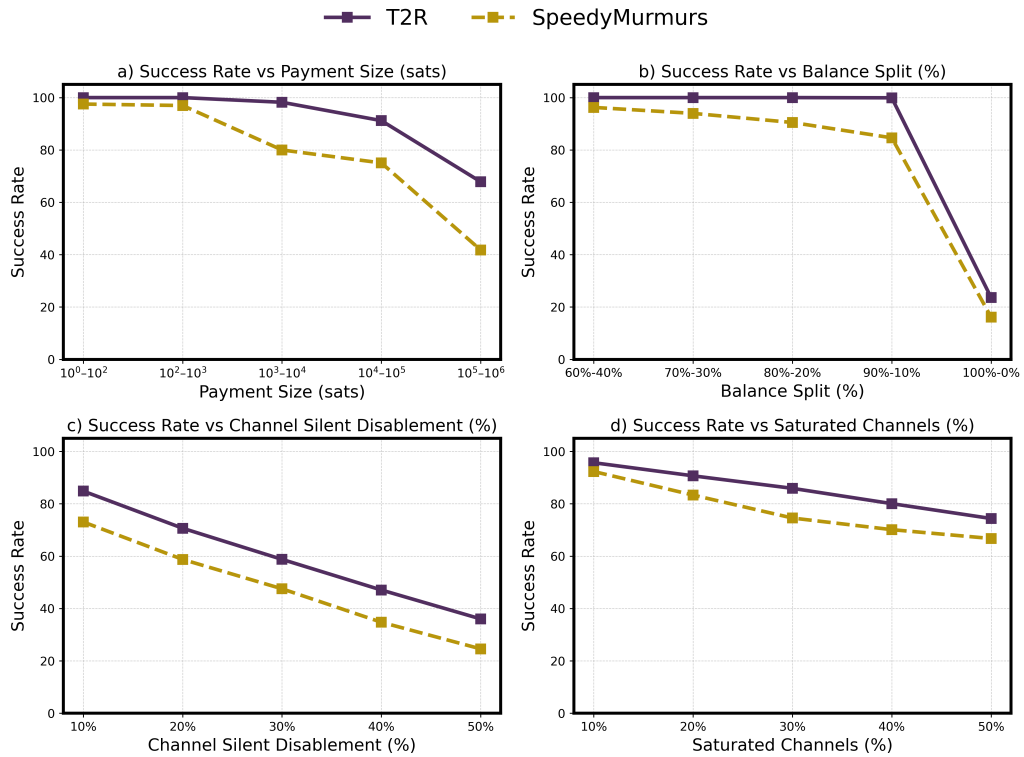

Figure 1 T2R vs. SpeedyMurmurs: success rate (higher is better) under four scenarios—payment size, balance split, silent channel shutdown, and random saturation.

Different initial balances.

T2R shows a higher success rate and a lower average hop per payment for each point of split capacity. Both approaches experience a notable drop in success rate as the split becomes extreme. While SpeedyMurmurs completes only a small fraction of payments under the 100:0 skew, T2R retains a modest advantage, successfully routing about one-quarter of payments by exploiting alternative branches. The hop count remains flat until the extreme imbalance, but still lower mean hops count and for the T2R .

Silent Disablement.

Across the different shutdown portions, T2R consistently outperforms the SpeedyMurmurs by *roughly twelve percentage points* in success probability. In every subplot, the T2R's curve has a meaningful gap with SpeedyMurmurs. Remarkably, it achieves this gain *while still trimming path length*: T2R's routes are 0.3–0.4 hops shorter compared to SpeedyMurmurs.

Random saturation.

T2R achieves higher success rate and a lower mean hop count than SpeedyMurmurs at every level of saturation:

- With only 10 percent saturated edge ($k = 10\%$) T2R holds steady at $\approx 95\%$ success, whereas SpeedyMurmurs slips below 92%.

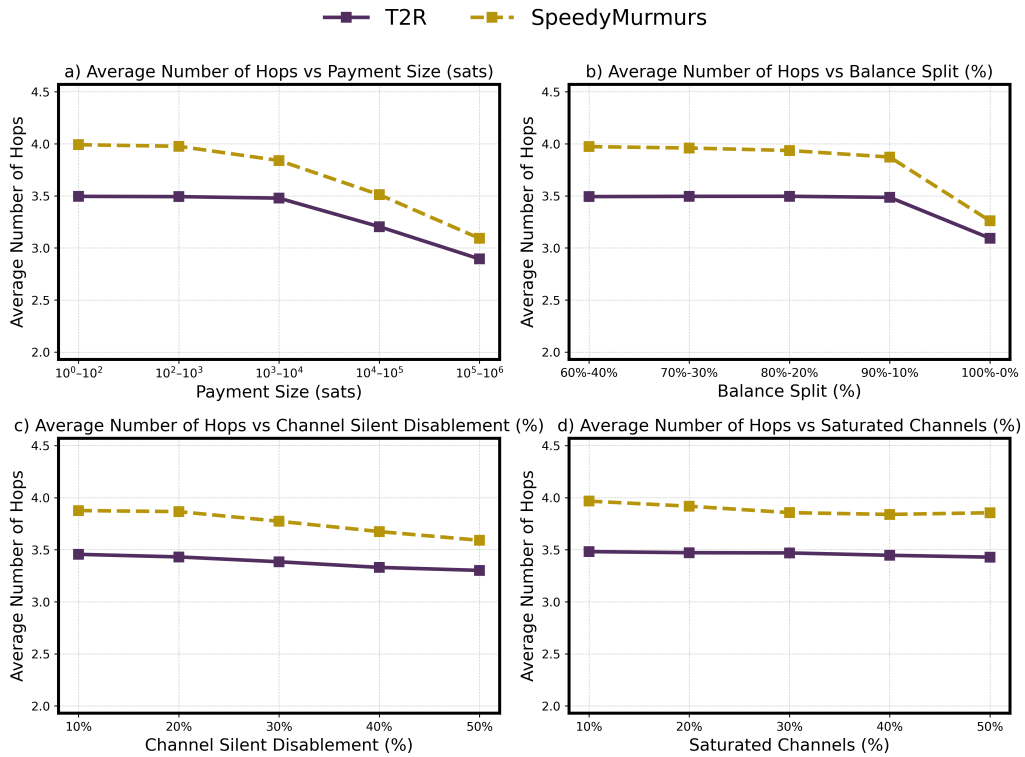

Figure 2 T2R vs. SpeedyMurmurs: average number of hops (lower is better) under four scenarios—payment size, balance split, silent channel shutdown, and random saturation.

- At the harshest setting ($k = 50\%$), both approaches experience a similar downward trend in success probability, with SpeedyMurmurs reaching around 66% and T2R retaining a lead at 74%, a difference of about eight percentage points.
- Across the entire sweep T2R's realized routes remain almost flat at $\bar{h} \simeq 3.4$ hops, consistently ~ 0.4 hops shorter than those of SpeedyMurmurs.

Overall, the two algorithms exhibit qualitatively similar behavior under stress, success rates degrade with increasing imbalance or saturation, but T2R consistently retains a modest advantage across all scenarios.

5.4 Ticket Size

T2R's overhead is dominated by the number of encrypted items inserted into its ticket. Hence, we profile its size under two workloads.

1. **$\mathbf{W_{all}}$: unrestricted pairs.** For the same 50 000 (s, d) pairs used in the main experiments, we build the candidate list for each node *without* capacity or HTLC limits and record $|B|$—the element count in the resulting ticket. Also, for each node, we select all viable pairs. We ignored all these constraints to reach the *absolute worst-case* scenario regarding the number of elements of the ticket, enabling us to provide an upper bound for the size of the ticket.
2. **$\mathbf{W_{long}}$: long-distance pairs.** To demonstrate the worst case, we generate a separate batch of 50 000 pairs with the shortest path length of ≥ 4 hops; the rest is identical.

Table 1 reports the key statistics (mean, median, 90$^{\text{th}}$ and 99$^{\text{th}}$ percentiles, and maximum) of the number of added elements in the ticket in both scenarios.

Table 1 Number of elements per ticket.

	Mean	Median	P_{90}	P_{99}	Max
W_{all}	51.0	11	126	611	2738
W_{long}	88.4	34	212	847	2997

- **Typical payments (W_{all}).** The median ticket carries just 11 elements, while 90 % of payments stay below 126 elements. With a Bloom filter configured for a target error $f = 10^{-5}$ (0.001 %), the 90$^{\text{th}}$-percentile ticket occupies only 377 B, whereas an equally secure GCS fits in 284 B
- **Long-distance payments (W_{long}).** Deliberately selecting endpoints at least four hops apart roughly doubles the mean element count to 88 and the median to 34. Even so, 90 % of these worst-case payments stay under 212 elements; at the same $f = 10^{-5}$ this translates to 634 B with a Bloom filter and 477 B with a GCS. Even the rare cases of outlier are controllable: it vanishes once we cap the *branch factor* K—that is, allow each node to contribute only a handful of its best channels to the ticket.

5.5 Effect of Ticket False Positives on Success Rate

T2R's reliability degrades only when an *authorized* next hop is a false positive: the intermediary node assumes an unusable edge is valid and finds no real edge left to forward the payment. To isolate this effect, we run same 50 000 payments from our main simulation under a perfect network (no disabled or saturated channels, balance split 50:50), and with balance update after each successful payment. For each payment, the sender inserts m authorized channel directions and creates a ticket sized optimally for the chosen false positive rate f. Payment amounts are drawn uniformly at random in the 100–1000 sat interval used elsewhere in the paper; a payment is declared *failed* if some intermediary node has no authorized next hop. Also, we simulate both single ticket and the TwinTicket.

5.5.1 Size of TwinTicket

We sweep the primary ticket's false–positive target $f \in \{10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$. The sender builds the TwinTicket by inserting *only* those node–pairs that collide in the primary ticket, meaning every pair that triggers a false positive in the primary ticket. Tables 2, 3 report the resulting element counts under both workloads from Section 5.4.

Table 2 Elements inserted into TwinTicket (τ_2), workload W_{all}.

False Positive f	Mean	Median	P_{90}	P_{99}	Max
10^{-2}	*40.6*	*29*	*92*	*176*	*467*
10^{-3}	*4.2*	*3*	*10*	*20*	*52*
10^{-4}	*0.4*	*0*	*1*	*3*	*10*
10^{-5}	*0.04*	*0*	*0*	*1*	*3*

Table 3 Elements inserted into TwinTicket (τ_2), workload W_{long}.

False Positive f	Mean	Median	P_{90}	P_{99}	Max
10^{-2}	*52.8*	*41*	*110*	*206*	*487*
10^{-3}	*5.3*	*4*	*12*	*22*	*52*
10^{-4}	*0.5*	*0*	*2*	*4*	*10*
10^{-5}	*0.05*	*0*	*0*	*1*	*3*

- **Typical payments (W_{all}).** For the TwinTicket setting we focus on three operating points that all achieve $\geq 99\,\%$ first-attempt success (Table 5 below, which is presented later in this section, illustrates the success rate based on false positives of the tickets) and compute their 90^{th}–percentile sizes under two encodings. We take $m_{90} = 126$ authorized pairs in the primary ticket obtained in Section 5.4.

  1. $(f_1, f_2) = (10^{-2}, 10^{-3})$, $m_1 = 126$, $m_2 = 92$:
     a. *Bloom Filter:* 150 B + 165 B = 315 B
     b. *GCS:* 126 B + 126 B = 252 B
  2. $(f_1, f_2) = (10^{-3}, 10^{-3})$, $m_1 = 126$, $m_2 = 10$:
     a. *Bloom Filter:* 226 B + 18 B = 246 B
     b. *GCS:* 174 B + 14 B = 175 B
  3. $(f_1, f_2) = (10^{-4}, 10^{-2})$, $m_1 = 126$, $m_2 = 1$:
     a. *Bloom Filter:* 301 B + 1 B = 302 B
     b. *GCS:* 237 B + 1 B = 238 B

  These figures confirm that even the most redundant ticket ($f_1 = 10^{-2}$) plus its TwinTicket remains well below half a kilobyte in Bloom form and under one-third of a kilobyte with a GCS.

- **Long-distance payments (W_{long}).** Taking the 90^{th}–percentile primary size $m_{90} = 212$ (Table 5) and the corresponding 90^{th}–percentile collision counts from Table 3, we obtain:

  1. $(f_1, f_2) = (10^{-2}, 10^{-3})$, $m_1 = 212$, $m_2 = 110$:
     a. *Bloom Filter:* 254 B + 197 B = 451 B
     b. *GCS:* 212 B + 152 B = 364 B
  2. $(10^{-3}, 10^{-3})$, $m_1 = 212$, $m_2 = 12$:
     a. *Bloom Filter:* 380 B + 22 B = 402 B
     b. *GCS:* 292 B + 12 B = 304 B
  3. $(10^{-4}, 10^{-2})$, $m_1 = 212$, $m_2 = 2$:
     a. *Bloom Filter:* 505 B + 3 B = 508 B
     b. *GCS:* 398 B + 2 B = 400 B

  Even in the long–path scenario the most redundant configuration remains below 0.6 kB (Bloom filter) and 0.4 kB (GCS) for 90% of payments. Also, same as single ticket scenario, capping the *branch factor* K—that is, allow each node to contribute only a handful of its best channels to the ticket will reduce size of the ticket for outliers.

5.5.2 Single Ticket vs Success Rate

We vary the ticket target error probability: $f \in \{10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$. Table 4 lists the success probability achieved at each tested false-positive target. We additionally calculate the size the tickets using the 90^{th}-percentile element counts reported in Section 5.4 using GCS. Once the filter is tightened to $f = 10^{-4} = 0.01\,\%$ the success rate already exceeds 90 %, and at $f = 10^{-5} = 0.001\,\%$ it climbs to $P_{\text{succ}} \approx 99.2\,\%$. Lowering f further adds only fractional improvement.

Table 4 Success rate versus filter error probability.

False-positive target f	Success probability [%]	Ticket size (Bytes)
10^{-2}	*15.0%*	*126B*
10^{-3}	*63.7%*	*173B*
10^{-4}	*93.5%*	*236B*
10^{-5}	*99.2%*	*283B*

5.5.3 TwinTicket vs Success Rate

To mitigate collisions we add TwinTicket. The sender constructs primary ticket with error f_1 as above, scans for collisions, and inserts those pairs into TwinTicket sized for an independent error f_2. Intermediate nodes accept a neighbor only if it is PRESENT in the primary ticket *and* ABSENT in the TwinTicket. We sweep the Cartesian grid $f_1, f_2 \in \{10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$ and report the resulting success matrix in Table 5. We also calculate the aggregate size of both tickets using the 90^{th}-percentile element counts for the reported in Section 5.4 using GCS and the numbers reported in W_{all}.

Table 5 Dual-filter success probability $P_{\text{succ}}(f_1, f_2)$ [%] (sum of the sizes of both tickets in bytes).

	f_2			
f_1	10^{-2}	10^{-3}	10^{-4}	10^{-5}
10^{-2}	*96.0% (218B)*	*99.7% (252B)*	*100% (298B)*	*100% (333B)*
10^{-3}	*97.9% (183B)*	*99.4% (187B)*	*100% (192B)*	*100% (195B)*
10^{-4}	*99.0% (237B)*	*99.8% (238B)*	*100% (238B)*	*100% (238B)*
10^{-5}	*99.8% (283B)*	*100% (283B)*	*100% (283B)*	*100% (283B)*

Table 5 demonstrates how quickly the second ticket drives the failures to zero. With the primary ticket set to $f_1 = 10^{-2} = 1\,\%$ the success rate already reaches 96 %; tightening f_2 beyond that yields no visible benefit. A practical sweet spot is therefore $(f_1, f_2) = (10^{-2}, 10^{-3})$: it already delivers $P_{\text{succ}} \geq 99.7\%$. Pushing either filter to 10^{-4} or 10^{-5} achieves a rise toward 100% but at a sharply higher bit cost than the primary ticket. Choosing $f_1 = 10^{-2}$ and $f_2 = 10^{-3}$ yields $P_{\text{succ}} > 99\,\%$ while the *sum* of the primary and TwinTicket is only ≈ 180 B.

6 Related Work

Early approaches.

The first proposals for payment-channel routing simply ran Dijkstra or A* on the public graph and probed candidate paths until one cleared liquidity. Because balances evolve with every transfer and are hidden from third parties, this "trial-and-error" strategy suffers a high failure rate and leaks balance information through repeated probes.

Lightning-specific schemes.

Research quickly turned to designs that respect Lightning's hop-by-hop HTLC semantics and privacy constraints. Flare [16] lets each node maintain topology within k hops and query the receiver's neighborhood; it is fast but leaks per-hop liquidity to the sender, as observed by SpeedyMurmurs' authors [17]. Multipart approaches try to tame hidden liquidity by splitting a payment. Spider [19] treats each fragment as a packet and applies congestion control, boosting aggregate throughput but stretching completion times. Flash [20] sends "mice" along cached shortest paths and reserves expensive max-flow computation for "elephants." Pickhardt–Richter flows [13] formalize the optimization as sequential min-cost-flow problems, achieving the highest known success probability at the expense of heavy sender-side computation. To assist lightweight wallets, introduces *trampoline payments* [5]: the sender routes to a hub that owns the global graph, and the hub computes the remainder of the route. Our three-ticket variant (§4.2) extends trampolines with fine-grained, privacy-preserving steering.

Credit-network lineage.

Off-chain credit networks such as Ripple and Stellar store bilateral IOUs on a ledger. Early Ripple relied on global consensus, exposing the entire path and amount on-chain. To avoid public state, SilentWhispers [12] used landmark nodes plus secure multiparty computation, guaranteeing privacy at high communication cost. SpeedyMurmurs [17] used metric-embedding trees as landmarks, optimizing latency and balancing load, but tree maintenance remains expensive and cannot encode sender blacklists. Our framework achieves the same privacy guarantee (no node learns non-incident edges) while discarding global trees entirely.

Approximate-membership filters.

AMQs are a standard tool for compact set representation: Bloom filters [4], counting Bloom filters [7], Cuckoo filters [6], quotient filters [2], and Golomb–Rice Coded Sets, adopted in Bitcoin for light-client block filters [15]. We leverage AMQs as "tickets" that travel with a payment and reveal only incident edges to each hop.

Position of our work.

Our T2R, locally steered source routing (i) sidesteps global trees, (ii) inherits incident-edge privacy from SpeedyMurmurs, and (iii) adds new control knobs—blacklists, receiver-issued tickets, trampoline segment steering—while keeping header overhead under one kilobyte and introducing no additional handshake round-trips.

7 Conclusion

This paper demonstrates that global landmarks or embeddings are *not* a prerequisite for privacy-preserving routing in payment-channel networks. By replacing them with a compact, cryptographically-keyed AMQ that *travels with the payment*, our **ticket-based framework** approach (i) removes continuous tree maintenance, (ii) keeps path privacy intact, and (iii) unlocks new control knobs—blacklisting misbehaving nodes, delegating route choices to receivers, and steering across trampoline hubs—without protocol round-trips.

In Lightning-sized topologies the TwinTicket configuration achieves $> 99\,\%$ first-try success at sub-kilobyte overhead, comfortably outperforming the SpeedyMurmurs baseline in both robustness and latency. Because ticket size scales linearly with the *number* (not length) of authorized channels, operators can trade header bytes for reliability on a per-payment basis.

References

1 ACINQ. Eclair. `https://github.com/ACINQ/eclair`, 2025. Version 0.9.0, accessed 27 May 2025.

2 Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. *Proc. VLDB Endow.*, 5(11):1627–1637, 2012.

3 Blockstream. Core lightning. `https://github.com/ElementsProject/lightning`, 2025. Version 24.02, accessed 27 May 2025.

4 Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

5 Christian Decker et al. Trampoline onion routing (draft proposal). `https://github.com/lightningnetwork/lightning-rfc/pull/829`, 2020. (accessed May 2025).

6 Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. Cuckoo filter: Practically better than bloom. In Aruna Seneviratne, Christophe Diot, Jim Kurose, Augustin Chaintreau, and Luigi Rizzo, editors, *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, CoNEXT 2014, Sydney, Australia, December 2-5, 2014*, pages 75–88. ACM, 2014.

7 Li Fan, Pei Cao, Jussara M. Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.

8 Florian Grötschla, Lioba Heimbach, Severin Richner, and Roger Wattenhofer. On the lifecycle of a lightning network payment channel. *CoRR*, abs/2409.15930, 2024.

9 Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 489–504. ACM, 2018.

10 Lightning Labs. BOLT #4: Onion routing protocol. https://github.com/lightning/bolts/blob/master/04-onion-routing.md.

11 Lightning Labs. lnd: Lightning network daemon. `https://github.com/lightningnetwork/lnd`, 2025. Version 0.18.0-beta, accessed 27 May 2025.

12 Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. Silentwhispers: Enforcing security and privacy in decentralized credit networks. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

13 Rene Pickhardt and Stefan Richter. Optimally reliable & cheap payment flows on the lightning network. *CoRR*, abs/2107.05322, 2021.

14 Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. Technical report, Self-published, 2016. Draft.

15 Jim Posen. Bip 158: Compact block filters for light clients. `https://github.com/bitcoin/bips/blob/master/bip-0158.mediawiki`, 2019.

16 Pavel Prihodko, Slava Zhigulin, Mykola Sahno, Aleksei Ostrovskiy, and Olaoluwa Osuntokun. Flare: An approach to routing in the lightning network. White paper, Bitfury Group Limited, July 2016.

17 Stefanie Roos, Pedro Moreno-Sanchez, Aniket Kate, and Ian Goldberg. Settling payments fast and private: Efficient decentralized routing for path-based transactions. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

18 Sindura Saraswathi and Christian Kümmerle. An exposition of pathfinding strategies within lightning network clients. *CoRR*, abs/2410.13784, 2024.

19 Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Kathleen Ruan, Parimarjan Negi, Lei Yang, Radhika Mittal, Giulia Fanti, and Mohammad Alizadeh. High throughput cryptocurrency routing in payment channel networks. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 777–796. USENIX Association, 2020.

20 Peng Wang, Hong Xu, Xin Jin, and Tao Wang. Flash: efficient dynamic routing for offchain networks. In Aziz Mohaisen and Zhi-Li Zhang, editors, *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies, CoNEXT 2019, Orlando, FL, USA, December 09-12, 2019*, pages 370–381. ACM, 2019.

21 Philipp Zabka, Klaus-Tycho Foerster, Stefan Schmid, and Christian Decker. Empirical evaluation of nodes and channels of the lightning network. *Pervasive Mob. Comput.*, 83:101584, 2022.