SPRITE: Secure and Private Routing in Payment Channel Networks[*]

Gaurav Panwar, Roopa Vishwanathan, George Torres, Satyajayant Misra

New Mexico State University

gpanwar@nmsu.edu, roopav@nmsu.edu, gtorresz@nmsu.edu, misra@nmsu.edu

Abstract

Payment channel networks are a promising solution to the scalability challenge of blockchains and are designed for significantly increased transaction throughput compared to the layer one blockchain. Since payment channel networks are essentially decentralized peer-to-peer networks, routing transactions is a fundamental challenge. Payment channel networks have some unique security and privacy requirements that make pathfinding challenging, for instance, network topology is not publicly known, and sender/receiver privacy should be preserved, in addition to providing atomicity guarantees for payments. In this paper, we present an efficient privacy-preserving routing protocol, SPRITE, for payment channel networks that supports concurrent transactions. By finding paths offline and processing transactions online, SPRITE can process transactions in just two rounds, which is more efficient compared to prior work. We evaluate SPRITE's performance using Lightning Network data and prove its security using the Universal Composability framework. In contrast to the current cutting-edge methods that achieve rapid transactions, our approach significantly reduces the message complexity of the system by 3 orders of magnitude while maintaining similar latencies.

1 Introduction

Researchers have been devising efficient techniques to make cryptocurrency transactions more scalable, e.g., Bitcoin currently processes around seven transactions per second, and Ethereum around thirty transactions per second [19, 6], compared to centralized payment systems, such as Visa Inc., which, at a conservative estimate, can support up to 1700 transactions per second [5]. For addressing this, Layer-2 protocols, such as payment channels have been proposed as a workaround [33, 38, 46, 47], where several thousands of transactions can be processed with minimal blockchain writes and with no changes required to the blockchain's underlying consensus mechanism (unlike other approaches such as sharding and alternate consensus mechanisms [31, 30, 39, 23, 42]). Payment channels also help enable *microtransactions*, which allow users to send small amounts of money, e.g., 10^{-4} Bitcoin, but without incurring high blockchain transaction fees [32].

[*]A short version of this work has been accepted to the 19th ACM ASIA Conference on Computer and Communications Security (ACM ASIACCS 2024). The authors thank Ryan Gentry and Alex Bosworth from Lightning Labs for their insights and discussion, which helped us improve the paper. The authors also thank Kartick Kolachala for his help with Table 1, and the anonymous reviewers for their feedback. This research was partially funded by the US National Science Foundation under grants #2148358 and #1914635, and the US Department of Energy grant #DE-SC0023392. Any opinions, findings and conclusions or recommendations expressed in this material are solely those of the authors and do not necessarily reflect the views of the US federal agencies.

Overview of payment channels: Two parties, Alice and Bob open a payment channel by depositing a certain amount of cryptocurrency into an address on a blockchain controlled by both parties' signing keys. Say, Alice deposits x coins, and Bob deposits y coins. Alice and Bob can conduct several transactions by exchanging authenticated messages, thus changing the distribution of the coins in the channel, but without writing anything to the blockchain. The net worth of the channel remains $x + y$ coins. At a mutually agreed-upon time, they can close the channel by writing a blockchain transaction that commits the final, authenticated distribution of the coins to the blockchain. The coins are paid to Alice and Bob per the final transaction. To facilitate transactions between two parties that may not have a payment channel currently open between them, decentralized payment channel networks (PCNs) that enable transitive payments have been proposed [36, 48, 37, 41], where two unconnected users can send/receive payments if there exists a path comprising of several users with payment channels between them.

Motivation: Layer-2 protocols such as PCNs are gaining widespread acceptance. Lightning Network, which is a popular PCN based on the Bitcoin blockchain, had over 6 million users and 28 million payment channels open between June 2021 to July 2022 [45, 33]. Peer-to-peer (p2p) transactions between users in PCNs are becoming increasingly common, e.g., in 2021, another popular PCN, Ripple, had 15 million unique p2p transactions annually, with a maximum path length of 43 hops [7, 4]. Routing protocols which help discover payment paths between sender and receiver are at the core of PCNs. There could exist several paths between a sender and receiver in a PCN with differing channel balances. Each hop on a path incurs a routing fee, hence longer paths cost more.

Routing in PCNs is fundamentally different from traditional network routing in both, intent and security/privacy requirements, hence network routing protocols cannot be trivially ported to PCNs. Assuming a network graph with nodes and weighted links connecting them, in regular network routing, the intent is to transmit data, not route payments. Transmitting data does not alter the state of the nodes, but routing payments do change each node's available link balances. In network routing, bandwidth capacities and router/switch identities are usually not considered private information, whereas, in PCNs, transaction amounts and node identities need to be kept private from all other nodes in the network. Transmission range and physical distance between devices are factors in network routing, but not in PCNs. PCNs reside entirely at the application layer, unlike network protocols in communication networks. Hence network routing protocols cannot be trivially ported.

Maximum flow algorithms such as Ford-Fulkerson [24] or Goldberg-Tarjan [25] would require either source routing or an external centralized, trusted entity to compute routes, besides having a high path computation overhead of $O(|V||E|^2)$ and $O(|V|^3)$ respectively, in a graph $G(V, E)$. In a decentralized network, nodes do not know the topology beyond their neighbors. While distributed versions of shortest path algorithms such as Dijkstra's algorithm exist [11, 12], they incur a computational complexity of $O(|V|^2) + O(|V|)$, which makes their scalability to large PCNs challenging.

Robust, scalable, decentralized PCN routing protocols hold the promise of making cryptocurrency transactions faster, hence, designing secure and efficient PCN routing protocols is a challenging research problem of practical significance.

Related Work: Several early PCN routing protocols were centralized where routing relied on trusted entities [36, 51, 40]. Some protocols did not support concurrency [36], while others chose paths without knowing whether the chosen path can satisfy a minimum asking amount [48]. Some routing protocols do source routing [37, 50] where a sender constructs the entire path from itself to the receiver, while many protocols do not consider security and privacy aspects [22, 57, 29, 26, 55, 21]. We provide a comparison of other relevant PCN routing protocols with SPRITE in Table 1, where our comparison metrics are informed by our security/privacy goals. The protocol in [41], while satisfying our three comparison metrics, has a very high communication overhead, where every transaction requires blockchain writes, which defeats the idea of off-chain PCNs. Real-world PCNs such as Lightning Network (LN) [33, 45, 13] implement a gossiping routing protocol, where each node gossips with its peers to build a local map

Table 1: Routing Protocols in PCNs

PCN Routing protocols	Privacy of nodes	Decentralized	Atomicity
FSTR [35]	✗	✗	✗
Eckey *et al.* [21]	✗	✓	✓
Auto tune [27]	✗	✗	✗
Kadry *et al.* [28]	✗	✗	✗
MPCN-RP [18]	✗	✗	✓
SilentWhispers [36]	✓	✗	✓
SpeedyMurmurs [48]	✓	✓	✗
BlAnC [41]	✓	✓	✓
Coinexpress [53]	✗	✓	✓
Vein [26]	✗	✗	✗
Spider [50]	✗	✗	✗
Flash [52]	✗	✗	✓
Robustpay [55]	✗	✗	✓
Robustpay+ [56]	✗	✗	✓
Webflow [54]	✓	✓	✗
SPRITE	✓	✓	✓

of the network. This has issues such as nodes not being able to validate information given by peers, and nodes often not finding the shortest path.

RobustPay+ [56] and its preliminary version Robustpay [55] focus on building a routing protocol for PCNs that constructs multiple paths from a sender to a receiver from which the sender chooses only one path to route the payment. MPCN-RP [18], builds a source routing protocol that minimizes the transaction fee, using a modified version of Dijkstra's algorithm, in which the length of the path is taken into consideration along with the edge weights. Auto-Tune [27] is a routing protocol that supports structured payments (transaction amount split into multiple pieces). All of these works [18, 27, 55, 56] do not take privacy of parties and privacy of network topology into consideration, do source routing, and do not support concurrent transactions.

Our Contributions: In this paper, we design a decentralized routing protocol for PCNs, SPRITE, which helps reduce trust assumptions, takes into account network dynamics, and preserves key security/privacy goals, while supporting concurrent transactions with short paths. We formally prove the security of SPRITE in the Universal Composability framework. We experimentally evaluate the performance of SPRITE using Lightning Network datasets and compare its performance with two other state-of-the-art schemes, on several network topologies. Our analysis shows that SPRITE performs significantly better over a wide array of quantitative and qualitative metrics while improving security and privacy.

Outline: In Section 2, we define our system and threat models, in Section 3, we give an overview of the workflow of SPRITE. In Section 4, we describe the protocols that constitute SPRITE. In Section 5, we give the security analysis of SPRITE. In Section 6, we describe our experiments, and in Section 7 we conclude the paper.

2 SPRITE System Model

In this section, we discuss the basics of a PCN, the parties involved in SPRITE and system parameters.

A PCN fundamentally can be conceptualized as a graph with users representing vertices and edges representing the payment channels between users. Figure 1 shows four parties and three two-party channels. The crossed-out number next to each party's name denotes that party's original balance in the channel, while the number above it denotes the new balance. The directionality of the arrows denotes the direction in which a payment can be processed.

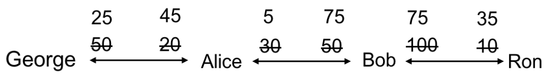


Figure 1: George sending 25 coins to Ron via two intermediaries Alice and Bob in a PCN.

2.1 Parties

1) **Routing nodes**: In SPRITE some nodes with high number of connections will serve as publicly identifiable *routing nodes* (RN), in exchange for a fee, and denote the set of RNs by $\mathbb{RN}$. RNs are already in use in real-world PCNs, such as Lightning Network as liquidity providers [34], we leverage them for routing. In SPRITE, RNs help facilitate transactions: broadly, we segment the path between the sender and the receiver, with each segment checkpointed by an RN. If RN_s and RN_r are the RNs closest to sender and receiver respectively, the payment from sender to receiver will progress as: sender $\to \mathrm{RN}_s \to \mathrm{RN}_1 \to \ldots \to \mathrm{RN}_n \to \mathrm{RN}_r \to$ receiver. The sender need only tell RN_s the identity of the destination RN_r, RN_s will find the shortest path to RN_r, who will, in turn, be contacted by the receiver. Consequently, node disconnections/failures or malicious activities on a segment are addressed and mitigated locally on each segment, and the rest of the path stays unaffected. Nodes volunteer to

be RNs, and RNs are financially incentivized to help route transactions. RNs periodically broadcast messages about the available liquidity on their links to nodes within a radius, $\mathrm{hopMax}_{\mathrm{RN}}$.

SPRITE does not require any special security assumptions on which entities can choose to be RNs, and accounts for malicious RNs in the system (discussed further in Section 2.4 and Section 5). In a given transaction, RNs involved do not know the identities of Alice, Bob, or any other nodes on the path (except intermediate RNs or their immediate neighbors). RNs do not have a privileged position from a monitoring standpoint, except RN_s and RN_r will know that somebody in their $\mathrm{hopMax}_{\mathrm{RN}}$ radius is the sender/receiver, respectively. Additionally, intermediate RNs will neither know the identities of, nor the distances to RN_s and RN_r for a given transaction. Alice and Bob are free to choose the RN_s and RN_r per transaction based on the RNs available in their respective *routingTable*s. If an Alice does not receive a broadcast message from any RN (indicating that she is outside the $\mathrm{hopMax}_{\mathrm{RN}}$ radius of all RNs in the system), she would need to connect either directly to a RN by forming a new payment channel or connect to another node in the network which is within $\mathrm{hopMax}_{\mathrm{RN}} - 1$ hops of some RN.

Since RNs are economically incentivized to facilitate transactions, we assume RNs will be online, but SPRITE's functioning will not be impacted by any specific RN(s) going offline. If an RN does go offline, the nodes depending on it for sending transactions will have to select other RNs.

2) **Perimeter nodes**: Perimeter nodes are nodes that are located closer to the boundary of an RN's broadcast area where the area is determined by radius $\mathrm{hopMax}_{\mathrm{RN}}$. The idea of using perimeter nodes is to enable RNs that are spaced across the network to be able to communicate, without having to establish direct connections with each other. Two RNs that are far apart and want to route a transaction just need to find a common perimeter node in their local routing tables, and can route payments using that node. Since we want to preserve the perimeter nodes' privacy from RNs, in SPRITE, perimeter nodes are only identified by nonces they generate. The perimeter nodes will send a unique nonce to any RN that they receive a broadcast message from. If two RNs receive the same nonce, then they know they can reach each other through the perimeter node that sent the nonce. RNs with overlapping neighborhoods may have several common perimeter nodes.

3) **Regular nodes**: Any node that is not a routing node or a perimeter node is a regular node. We assume all nodes are rational and will act in their best economic interests. We assume the sender and receiver in a transaction can exchange messages out-of-band with each other, but payments are routed through nodes on the PCN. We use the terms users and nodes interchangeably.

4) **Blockchain**: SPRITE can work with any permission-less blockchain, and does not rely on blockchain-specific constructs such as hash time lock contracts (HTLCs) used in the Bitcoin blockchain, or smart contracts which are supported only by Turing complete blockchains, such as Ethereum. The blockchain is only used for opening/closing payment channels, thus avoiding excessive write/validator fees.

2.2 Setup

When a node joins the PCN, it establishes payment channels with other nodes who offer to connect with it or accept its connection offer. A node needs to connect to at least one other node to be part of the PCN. Nodes only reveal their identities to peers that they share a channel with. In this paper, we refer to peers sharing a channel as neighbors. Every node's identity is represented by a keypair denoted by $(\mathrm{VK}_i, \mathrm{SK}_i)$, of which VK_i is revealed to its neighbors. RNs will need to make their identities, i.e., verification keys, known to all nodes in the PCN, so nodes can use them for routing transactions.

Cryptographic Primitives: A *sequential aggregate signature* is a cryptographic primitive in which a series of users sign a message, where the final signature is computed sequentially by each user who adds her signature on her message. We use sequential aggregate signatures [44] (defined in Appendix 8.1) to maintain the privacy of non-RN nodes in the network (no need for publicly registered signing keypair) while still allowing for authentication of broadcast messages during the bootstrap phase. Furthermore, this helps from an efficiency perspective, since only one final signature needs to be verified rather than a series of signatures.

2.3 System Parameters

Transactions in a PCN might on occasion fail, e.g., due to abrupt node disconnections and insufficient liquidity along a path, thus necessitating retries. We set the number of times a transaction can be retried after a failure as a system-wide parameter, $\mathsf{maxRetries}$. We also assume that each node i maintains a local state where it stores the number of times each transaction is retried, specifically, it maintains an arithmetic counter for each transaction $(txid_i)$, $\mathsf{retry}.txid_i \in \mathbb{Z}^+, i \in \mathbb{Z}^+$. If $\mathsf{retry}.txid_i == \mathsf{maxRetries}$, any new messages about that transaction will be rejected so the transaction can be tried on other paths. After transaction $txid_i$ has been completed, times out, or is revoked, $\mathsf{retry}.txid_i$ is deleted.

Hops: We define five parameters used in SPRITE: $\mathrm{hopMax}_{\mathrm{RN}}$, hopMax, $\mathrm{pathStretch}$, $\mathrm{hopCount}$, and $\mathrm{hopBand}$. $\mathrm{hopMax}_{\mathrm{RN}}$ is the maximum number of hops an RN's broadcast message travels, hence defining the RN's neighborhood. $\mathrm{hopBand}$ is used for determining the distance of perimeter nodes. For example, if node r is an RN, $\mathrm{hopMax}_{\mathrm{RN}}$ is set to 20 hops and $\mathrm{hopBand}$ is set to 3, then all nodes that are at 18, 19, and 20 hops away from node r act as perimeter nodes. $\mathrm{hopMax}_{\mathrm{RN}}$ and $\mathrm{hopBand}$ are set individually by RNs. hopMax is a dynamic parameter that denotes the maximum number of hops a transaction can travel in a given segment. It is set by the sender for a given segment based on the estimated $\mathrm{hopCount}$ in the sender's *routingTable*. $\mathrm{pathStretch}$, set by the sender, denotes an absolute upper bound on hopMax and is intended to be used only in case of routing problems that call for transaction retries within a segment. $\mathrm{hopCount}$ at a given node denotes the number of hops traveled by a message up until that node.

Timers: Transactions in SPRITE have two phases, *hold* and *pay*, and their corresponding segment-specific timers, $te_1.txid$ and $te_2.txid$, are maintained by each node participating in a given transaction designated by $txid$. These are internal countdown timers that are maintained by each node locally and are used by the nodes individually to determine when they should timeout the given transaction and retry on a different path. Since each segment in the *hold* phase terminates at an RN, timer te_1 is cleared by nodes in a segment after a successful *hold* phase when the receiving downstream RN responds with an acknowledgment message for the transaction. Else, nodes will retry the transaction's *hold* phase on another path in the given segment after te_1 expires. Timer te_2 is cleared by all nodes in a transaction segment after a successful *pay* phase when they receive an acknowledgment that the payment has concluded successfully in their segment. Else, if te_2 expires, then the transaction is retried for *hold* and *pay* phases in the given segment. In SPRITE, we consider te_1 and te_2 to be system parameters set based on current network statistics and dynamics.

Fees: Similar to prior works, we assume RNs get paid a fixed amount periodically, contributed to by other nodes, and do not impose routing fees for transactions.[1] An economic analysis of routing fee models and optimal routing fee design is an orthogonal problem.

2.4 Threat Model and Security/Privacy Goals

Adversary actions: An adversary can adaptively corrupt any subset of users, including regular nodes, perimeter nodes and RNs, upon which the corrupted nodes' channels will be controlled by the adversary. The adversary can then cause the corrupted users to behave in arbitrarily malicious ways, including misrouting payments and/or disseminating false information. We do not consider any node dropping/ignoring routing requests as malicious behavior, since that just means the node does not wish to participate in the given transaction, and a path that does not involve that node has to be found.

[1]In real-world PCNs such as LN, routing nodes currently get paid the same as other nodes, although there are proposals to update the fee structure [9, 8, 10].

Adversary goals: An adversary wants to know nodes' identities that are not its immediate neighbors, including sender/receiver identities, and/or make people lose money, i.e., violate the atomicity of transactions.

Privacy-preservation: No node, not even RNs, know the identities of the sender, receiver, or any non-RN intermediaries for routing transactions, thus preserving sender and receiver privacy. SPRITE does not require the topology of the network to be known by any participating node in the system, as is standard in topology-hiding PCNs.[2]We assume the adversary cannot corrupt *all* users in the PCN.

Security/Privacy goals:

1) *Privacy of nodes*: Nodes should not know the identities of any nodes beyond their neighbors and RNs, nor garner any information (number of channels or balances) about other nodes.

2) *Transaction privacy*: No node should know the identities of the sender, receiver or the intermediaries in a transaction, unless it shares a channel with them. It should also not know amounts transferred in transaction paths it is not a part of.

3) Atomicity: Either a payment goes through in its entirety or not at all, i.e., either all link weights along a transaction path get updated by the transaction amount or none at all. In other words, no honest party should lose credits because of the malicious behavior of other parties in the network.

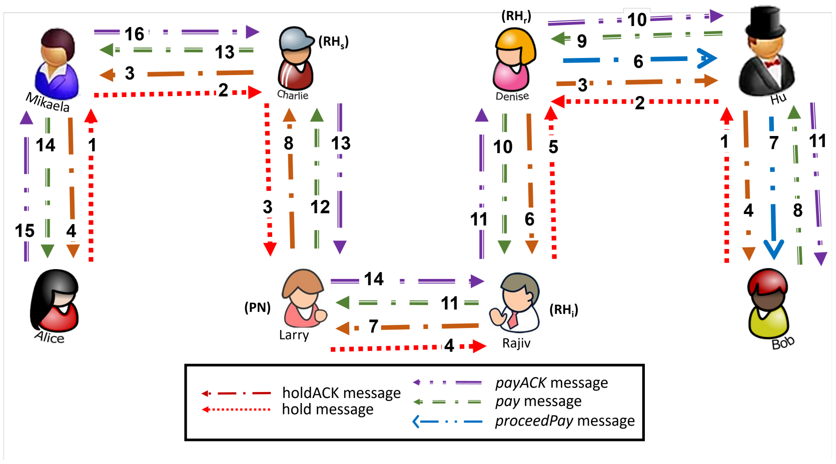


Figure 2: Example of SPRITE protocol

3 Example run through of SPRITE

In this section, we provide an example run-through of the SPRITE protocol using Figure 2 where Alice is the sender and Bob is the receiver for a transaction. For presentation clarity, we do not pictorially depict multiple intermediary nodes between each of the parties in Figure 2, but there exist multiple nodes between each of the depicted parties. The bootstrap phase is used by RNs in the system to broadcast update messages that help nodes in their vicinity build routing tables. At the end of the broadcast phase, each node in the network will have a local routing table that indicates which RNs are reachable and through which of the node's neighbors. The nodes' routing tables also have estimates about the hop count and liquidity available to the corresponding RNs. The bootstrap phase also allows RNs to obtain

[2]In LN, although edited snippets of the topology are made available for research purposes [20], one cannot extract the full network topology, as nodes' channel balances are not made public. Further, each payment channel funding transaction is a Pay-to-Witness-Script-Hash (P2WSH) address, and the nature of the script (a 2-of-2 multisig) will only be revealed once the funding transaction output is spent. Even if this were known/guessed, not all 2-of-2 multisig scripts on the Bitcoin blockchain correspond to payment channels. Finally, signing/verification keys are rotated by nodes for every channel (see [13]).

information about what RNs are in their adjacent neighborhoods, the perimeter nodes that connect them, and how to reach RNs that are not in the adjacent neighborhoods.

When a transaction needs to occur, Alice and Bob coordinate out-of-band to confirm their closest reachable RNs (Charlie for Alice and Denise for Bob). From their routing table estimates, Alice and Bob decide the transaction amount based on the estimated liquidity available between Alice-Charlie and Denise-Bob, according to Alice's and Bob's corresponding routing tables. Alice sends a $hold_s$ message to Charlie via one of her neighbors and this message is passed on by each node along the path including Mikaela, until it reaches Charlie (Figure 2, Steps 1-2 on Alice-Charlie segment). Simultaneously, Bob sends a $hold_r$ message towards Denise through Hu (Figure 2, Steps 1-2 on Bob-Denise segment). Along the path, all nodes create pair-wise multisig hold contracts with their neighbors to reserve the transaction amount and set some local variables including hold phase timer (te_1) and pay phase timer (te_2).

When the messages reach Charlie and Denise, they reply with *holdACK* messages so that all nodes receiving the *holdACK* message clear their local te_1 timers and will no longer timeout and retry another path (Figure 2, Steps 3-4 in Alice-Charlie segment and Bob-Denise segment). Additionally, Charlie updates Alice's message so that it can be routed within the network through any intermediate RNs (Rajiv in this case) and is finally received by Denise (Figure 2, Steps 3-5 on Charlie-Rajiv and Rajiv-Denise segments). The message is updated by each perimeter node (Larry) and RN (Rajiv) on the path to facilitate forwarding the message towards Denise.

All nodes on the Alice to Denise path also set corresponding te_1 and te_2 timers (Figure 2, Steps 3-4 in Charlie-Rajiv segment and Step 5 in Rajiv-Denise segment) which are cleared when the corresponding RN in that segment is reached (Figure 2, Steps 5-6 in Charlie-Rajiv segment and Step 6 in Rajiv-Denise segment). In Figure 2, all nodes between Charlie and intermediate RN Rajiv, including perimeter node Larry will clear their te_1 timers after they receive a *holdACK* from Rajiv (Figure 2, Steps 6-8 on Charlie-Rajiv segment) and nodes between Rajiv and Denise will clear their timers when *holdACK* from Denise is received (Figure 2, Steps 6 on Rajiv-Denise segment). When Denise receives the two $hold_r$ and $hold_s$ messages, she sends Bob a *proceedPay* message (Figure 2, Steps 6-7 on Bob-Denise segment). On receiving *proceedPay*, Bob creates a *pay* message and sends it towards Denise (Steps 8-9 on Bob-Denise segment), which is then forwarded towards Charlie through intermediate RNs (Steps 10-12 on Charlie-Rajiv and Rajiv-Denise segment), and finally to Alice (Steps 13-14 on Alice-Charlie segment).

Each RN on the path replies with a *payACK* message when it receives a *pay* message and thus clearing timer te_2 for all nodes receiving the *payACK* message (Denise's *payACK* represented by Steps 10-11 on Bob-Denise segment, Rajiv's *payACK* represented by Step 11 on Rajiv-Denise segment, and Charlie's *payACK* represented by Steps 13-14 on Charlie-Rajiv segment). Finally, Alice sends out her own *payACK* when she receives the *pay* message (Steps 15-16 on Alice-Charlie segment), clearing the te_2 timers for nodes in the last segment, and effectively concluding the transaction.

4 Construction of SPRITE

In the current Lightning Network, most new nodes connect to highly connected nodes in the network. This leads to a high concentration of nodes connected directly or with low hopcounts to well-connected nodes (RNs). This setup does not provide sender/receiver privacy from the highly connected nodes and there is the danger of highly connected nodes' link balances getting depleted quickly. Furthermore, if any RNs get disconnected or go offline, many other nodes would get disconnected from the network. In a network similar to Lightning, where RN nodes are closely located in terms of hop count, an RN-to-RN broadcast algorithm, which we refer to as R2RB (Algorithm 9) and define in Appendix 8.2, would work well.

However, if a PCN is built from the ground up with the security of the transactions and privacy of the nodes as the focus, it is easy to assert that nodes would not necessarily always set up payment channels directly with well-known nodes (RNs) in the network since this would make the RN their next-hop neighbor and thus leak their identity as well as all their transactions' details. In a truly distributed network, new nodes would join other nodes in the periphery that they trust and not just RNs. In a system where RNs are located further apart, R2RB suffers from high message complexity due to long distances for *RN-Update* broadcast messages. We developed Algorithms 1, 2, henceforth referred to as R2NB, which reduces the distance each RN broadcasts to during the Setup phase, thus reducing the message complexity and adding to the efficiency of our scheme. The *hold* and *pay* phases remain the same for both approaches.

In practice, the first bootstrap phase in a given PCN will involve tuning of the $\mathrm{hopMax}_{\mathrm{RN}}$ parameter by the RNs to get an optimal overlap of perimeter nodes between neighboring RNs. The $\mathrm{hopMax}_{\mathrm{RN}}$ parameter is only used during the bootstrap phase of SPRITE and helps in limiting the number of broadcast messages from each RN; it is not used during a transaction. When new nodes join the network, they will receive their neighbors' *routingTable*s regardless of their distance from any given RN and thus will join the neighborhood of the RN(s) that their neighbors occupy.

Algorithm 1: R2NB: Bootstrap broadcast from RN to perimeter nodes

```
1  Each node $i$ initializes a table, $routingTable_i$ containing columns:
     (reachable RNs, next hop neighbor $j$, $\mathrm{currMax}_s$, $\mathrm{currMax}_r$, hopCount, $t_e$).
2  for each RN, $k \in \mathbb{RN}$ do
3      $k$ does $\mathsf{AS.Setup}(1^\lambda) \to pp_k$ and runs $\mathsf{AS.KeyGen}(pp_k) \to (\mathrm{sk}_k, \mathrm{vk}_k)$.
4      Create a tuple $m_k = (\textit{RN-Update}, pp_k, \mathrm{VK}_k, \mathrm{currMax}^k_s, \mathrm{currMax}^k_r, \mathrm{hopCount} = 0,$
         $\mathrm{hopBand}, \mathrm{hopMax}_{RN}, \mathsf{ts})$ for each neighbor $j$, $j \in [1..l]$ where $l$ is the total number of
         neighbors of $k$. Create $\sigma'_k \leftarrow \mathsf{Sign}(\mathrm{SK}_k, m_k)$ and set $m'_k = (m_k, \sigma'_k)$ Create signature
         $\sigma_k \leftarrow \mathsf{AS.Sign}(\mathrm{sk}_k, \perp, \perp, \perp, m'_k)$.
5      return $M = ((m'_k), (\mathrm{vk}_k), \sigma_k)$ to each neighbor $j$.
6  for each node $i$ in the network on receiving an RN-Update message from neighbor $j$ do
7      On receiving $M = ((m'_k, \ldots, m_j), (\mathrm{vk}_k, \ldots, \mathrm{vk}_j), \sigma_j)$, $i$ parses $(m_k, \sigma'_k) \leftarrow m'_k$ and
         $(\textit{RN-Update}, pp_k, \mathrm{VK}_k, \mathrm{currMax}^k_s, \mathrm{currMax}^k_r, \mathrm{hopCount}, \mathrm{hopBand}, \mathrm{hopMax}_{RN}, \mathsf{ts})$
         $\leftarrow m_k$.
8      if ($\mathsf{Verify}(m_k, \mathrm{VK}_k, \sigma'_k) \to 0) \vee (\mathsf{AS.Verify}((m'_k, \ldots, m_j), (\mathrm{vk}_k, \ldots, \mathrm{vk}_j), \sigma_j) \to 0)$ then
9          Return $\perp$.
10     $i$ checks that hopCount value in all messages $(m'_k, \ldots, m_j)$ are incremented by 1 in each
         message. If not, return $\perp$.
11     $i$ runs $\mathsf{AS.KeyGen}(pp_k) \to (\mathrm{sk}_i, \mathrm{vk}_i)$.
12     $i$ updates its local routingTable for RN $k$ and neighbor $j$ by updating the expiry time
         $t_e = \mathrm{currTime} + e$, $\mathrm{currMax}^k_s$, and $\mathrm{currMax}^k_r$.
13     if (($\mathrm{hopMax}_{RN}$ $-$hopBand $<$ hopCount) $\wedge$ (hopCount $\leq \mathrm{hopMax}_{RN}$)) then
14         Create a nonce $\mathrm{Nonce}_i \leftarrow\!\!\$ \{0,1\}^\lambda$.
15         Create return message $mr'_i$ by updating contents of $m_j$ as $mr'_i$ = (RN-UpdateReply, $\cdot$, $\cdot$,
             $\mathrm{currMax}^k_s$, $\mathrm{currMax}^k_r$, hopCount, $\cdot$, $\cdot$, $\cdot$,$\mathrm{Nonce}_i$) where hopCount = hopCount + 1,
             $\mathrm{currMax}^k_s = min(\mathrm{currMax}^k_s, \mathrm{lw}_{j,i})$, and $\mathrm{currMax}^k_r = min(\mathrm{currMax}^k_r, \mathrm{lw}_{i,j})$.
16         $i$ creates signature $\sigma_i \leftarrow \mathsf{AS.Sign}(\mathrm{sk}_i, \sigma_j, (m'_k, \ldots, m_j), (\mathrm{vk}_k, \ldots, \mathrm{vk}_j), mr'_i)$. $i$ sends
             $MR = ((m'_k, \ldots, m_j, mr'_i), (\mathrm{vk}_k, \ldots, \mathrm{vk}_j, \mathrm{vk}_i), \sigma_i)$ back to neighbor $j$.
```

4.1 Bootstrap phase

This phase is described in Algorithm 1 and Algorithm 2. In the bootstrap phase the RNs first broadcast messages in the PCN within hopMax_{RN} hops, advertising their available liquidity. The goal is to make nodes within hopMax_{RN} aware that they can reach the respective RN, and help them construct their local routing tables.

RN **broadcast to bootstrap neighborhood (Algorithm 1):** In Algorithm 1, Lines 2-5, each RN k sets up the public parameters of an aggregate signature scheme, pp_k and creates an aggregate signature keypair for itself $(\mathrm{sk}_k, \mathrm{vk}_k)$. This is so all nodes in the RNs neighborhood can set up pseudonymous keypairs to hide their identity while propagating messages. It then composes an update message m_k, to be sent to all its neighbors. The message m_k contains k's available liquidity in the outgoing direction, $\mathrm{currMax}_s^k$, liquidity in the incoming direction $\mathrm{currMax}_r^k$, and its real identity, VK_k. It also sets $\mathrm{hopCount}$ to be zero and sets the $\mathrm{hopBand}$. Perimeter nodes will be the farthest nodes from k in the band defined by

Algorithm 1: R2NB: Bootstrap broadcast from RN to perimeter nodes (continued)

```
17      if $\mathrm{hopCount} \geq \mathrm{hopMax}_{RN}$ then
18          Return $\perp$.
19      for each neighbor $s$ do
20          $i$ creates $m_i$ by updating contents of $m_j$ as $\mathrm{hopCount} = \mathrm{hopCount} + 1$,
              $\mathrm{currMax}_s^k = min(\mathrm{currMax}_s^k, \mathrm{lw}_{i,s})$, and $\mathrm{currMax}_r^k = min(\mathrm{currMax}_r^k, \mathrm{lw}_{s,i})$.
21          $i$ creates signature $\sigma_i \leftarrow \mathsf{AS.Sign}(\mathrm{sk}_i, \sigma_j, (m'_k, \ldots, m_j), (\mathrm{vk}_k, \ldots, \mathrm{vk}_j), m_i)$.
22          $i$ sets $M = ((m'_k, \ldots, m_j, m_i), (\mathrm{vk}_k, \ldots, \mathrm{vk}_j, \mathrm{vk}_i), \sigma_i)$ and return $M$ to neighbor $s$.
23  for each node $j$ in the network on receiving an RN-UpdateReply message from neighbor $o$ do
24      On receiving $MR = ((m'_k, \ldots, mr'_i, \ldots, mr'_o), (\mathrm{vk}_k, \cdots, \mathrm{vk}_i, \ldots, \mathrm{vk}_o), \sigma_o)$.
25      if ($\mathsf{AS.Verify}((m'_k, \ldots, mr'_o), (\mathrm{vk}_k, \ldots, \mathrm{vk}_o), \sigma_o) \to 0$) then
26          return $\perp$.
27      if $j$ is the RN $k$ then
28          if $(\mathrm{hopMax}_{RN} - \mathrm{hopBand}) \overset{?}{\leq} |\{m'_k, \ldots, mr'_i, \ldots, mr'_o\}|/2 \overset{?}{\leq} \mathrm{hopMax}_{RN})$ then
29              Add $(\mathrm{Nonce}_i, o, \cdot, \cdot, \cdot, \cdot)$ to $RNroutingTable_k$.
30      else
31          Add $\mathrm{Nonce}_i$ and neighbor $o$ to routingTable.
32          Update contents of $mr'_o$ as $mr'_j$ = (RN-UpdateReply, $\cdot$, $\cdot$, $\mathrm{currMax}_s^k$, $\mathrm{currMax}_r^k$,
              $\mathrm{hopCount}$, $\cdot$, $\cdot$, $\cdot$,$\mathrm{Nonce}_i$) where $\mathrm{hopCount} = \mathrm{hopCount} - 1$,
              $\mathrm{currMax}_s^k = min(\mathrm{currMax}_s^k, \mathrm{lw}_{j,o})$, and $\mathrm{currMax}_r^k = min(\mathrm{currMax}_r^k, \mathrm{lw}_{o,j})$.
33          $j$ creates signature $\sigma_j \leftarrow \mathsf{AS.Sign}(\mathrm{sk}_j, \sigma_o, (m'_k, \ldots, mr'_o), (\mathrm{vk}_k, \ldots, \mathrm{vk}_o), mr'_j)$.
34          Forward message $MR = ((m'_k, \ldots, mr'_o, mr'_j), (\mathrm{vk}_k, \cdots, \mathrm{vk}_o, \mathrm{vk}_j), \sigma_j)$ to neighbor
              from who RN-Update message of $k$ with timestamp ts was received.
```

nodes lying between hopMax_{RN} hops and $(\mathrm{hopMax}_{RN} - \mathrm{hopBand})$ hops from k. Each RN k can set its $\mathrm{hopBand}$ independently. RN k timestamps and signs the message m_k using the signing key tied into its real identity, and produces a signature, σ'_k. It then again signs σ'_k and m_k using its aggregate signature signing key and creates an aggregate signature, σ_k, which is sent to k's neighbors.

In Line 6, each node i within hopMax_{RN} receives a set of messages $(m'_k, \ldots, m_j)$ and a set of verification keys $(\mathrm{vk}_k, \ldots, \mathrm{vk}_j)$ and a single aggregate signature σ_j which represents the aggregate signature of all nodes along the path from RN k to node j. Node i will then verify the signature, perform other checks (Lines 6-10), and update the values of $\mathrm{currMax}_s^k$ and $\mathrm{currMax}_r^k$ in its local routing table (Line 12). If node i is a non-perimeter and non-RN node, it then composes a new *RN-Update* message to forward to its neighbors. It increments the $\mathrm{hopCount}$ by one, computes the new values of $\mathrm{currMax}_s^k$, $\mathrm{currMax}_r^k$ based on its local channel balances, appends its message to the message list and generates an aggregate signature on the appended list. It then sends the updated *RN-Update* message to its neighbors (Lines 19-22).

If node i happens to be a perimeter node (Lines 13-16) based on the hopCount of the received message, it generates a nonce Nonce_i. It creates an *RN-UpdateReply* tuple that includes Nonce_i, updated values of currMax_s^k, currMax_r^k, and hopCount, and sends it to it's previous node towards RN. All perimeter nodes also forward the *RN-Update* message until it reaches the node at hopMax_{RN} hops, who will send a reply but not broadcast the message further.

When nodes receive an *RN-UpdateReply* tuple, they act differently depending on whether they are an RN or a regular node. If the receiving node is an RN, then the message has traveled to the perimeter nodes and back. The receiving RN will store the reply information sent by the perimeter nodes in its *RNroutingTable*, indexed by the Nonce value sent by the perimeter node (Line 27-29). On the other hand, if the node receiving *RN-UpdateReply* is a non-RN node, then it updates its local routing table again with the received information (currMax_s^k, etc.), adds the perimeter node's nonce to its local routing table, computes new values of currMax_s^k, currMax_r^k, decrements hopCount, and sends the signed message to the neighbor from whom it received the corresponding *RN-Update* (Lines 31-34). Here e is the system-wide parameter for depicting the time duration after which a record is considered expired/stale in nodes' *routingTable*. In case node i had received the same message tuple with a lower hopCount earlier, it drops the message to avoid loops. A possible optimization is nodes updating currMax_s^k, currMax_r^k only once, instead of twice, i.e., on receipt of the *RN-Update* tuple (Line 20) and not again after receipt of the *RN-UpdateReply* tuple (Line 32). New nodes joining the PCN get *routingTable*s from their neighbors as

Algorithm 2: RNs exchanging nonces

```
1 Each $\text{RN}_i \in \mathbb{RN}$ creates a table with rows $(\text{Nonce}_m, j, \cdot, \cdot, \cdot, \cdot)$, where $j$ is the neighbor $\text{RN}_i$
    received $\text{Nonce}_m$ from. Let $\mathbb{N}_i$ be the set of all nonces obtained by $\text{RN}_i$.
2 Each $\text{RN}_i$ then picks $\alpha \leftarrow\$ \mathbb{Z}_p$, picks $d \in \mathbb{Z}^+$, and creates set
    $\mathcal{R}_i = \{r_i, \forall\, i \in [1..d]; r_i \leftarrow\$ \{0,1\}^\lambda\}$, $d = |\mathcal{R}_i|$. $\text{RN}_i$ then sets $N_i = \mathbb{N}_i \cup \mathcal{R}_i$.
3 $\text{RN}_i$ sends $\mathbb{N}_i$ to all $\text{RN}_j \in \mathbb{RN} \setminus \text{RN}_i$.
4 Each $\text{RN}_i$ computes $\mathbb{N}_{ij} \leftarrow \mathbb{N}_i \cap \mathbb{N}_j$ for all $\text{RN}_j \in \mathbb{RN} \setminus \text{RN}_i$, and builds its *RNroutingTable*
    locally.
```

soon as they join and will participate in *RN-Update* broadcasts in the next time epoch. No re-calculation or broadcasts happen when new nodes join the network. For highly dynamic networks, the epoch value can be tuned or lowered so that the *RN-Update* broadcast messages account for significant changes in the topology. The cost of the *RN-Update* bootstrap phase is similar across epochs and depends on the current size of the network during the broadcast.

Algorithm 3: Alice-RN_s - $\cdots$ - RN_r hold segments

```
1 Alice picks $\text{RN}_s$ and Bob picks $\text{RN}_r$. Bob sets *preimage* $\leftarrow\$ \{0,1\}^\lambda$ and
    *digest* $= H($*preimage*$)$, and shares *digest* with Alice.
2 Let $\nu$ be the amount of credits Alice wishes to send to $\text{RN}_s$. Alice picks
    $token, preimage_{txid} \leftarrow\$ \{0,1\}^\lambda$, $txid = H(preimage_{txid})$, and sends $txid$ to Bob.
3 Alice does $C_{\text{RN}_r} = E_{PK_{\text{RN}_r}}(token, \nu, txid)$ and $C_{\text{RN}_s} = E_{PK_{\text{RN}_s}}(VK_{\text{RN}_r}, \nu, txid, C_{\text{RN}_r})$.
4 Alice looks up her *routingTable* and picks a tuple $(\text{RN}_s, \text{node}_k, p_k)$, with
    $p_k = (\text{hopCount}, \text{currMax}_s, \text{currMax}_r, t_e)$ where $\text{currMax}_s \geq \nu$ and sets
    $\text{hopMax} = \text{hopCount} + \text{pathStretch}$. Alice creates a tuple $(hold_s, \text{RN}_s, VK_{\text{RN}_s}, \nu, txid,$
    $C_{\text{RN}_s}, \text{hopMax}, digest, te_1, te_2)$ and sends it to $\text{node}_k$.
5 for Each node ($\text{node}_i$) in the network do
6     Follow Algorithm 5
```

RNs exchanging nonces (Algorithm 2): After the PCN is bootstrapped, the RNs need to setup their local *RNroutingTable*s which will help them find other RNs. At the end of Algorithm 1, each RN i would have received *RN-UpdateReply* tuples of the form $(\text{Nonce}_m, \cdot, \cdot, \cdot, \cdot, \cdot)$ from its neighbors, where m is a perimeter node within i's hopMax_{RN} radius. RN i will receive several tuples containing nonces, we represent the set of unique nonces that i receives by $\mathbb{N}_i$ (Line 1). RN i then pads the set $\mathbb{N}_i$ with random strings and generates a larger set $\mathcal{R}_i$ (Line 2). This is to ensure that other RNs cannot guess the size of $\mathbb{N}_i$, thus preserving privacy. SPRITE not only hides the identity of the perimeter nodes against all RNs in the system using the randomly generated nonces by perimeter nodes, but also hides the number of perimeter nodes each RN has within its hopMax_{RN} radius. All RNs then exchange their nonce sets and each RN finds the intersection of its set with other RNs' sets (Line 4). If one wants to hide even the nonce values, we can use more involved protocols such as private set intersection [43].

Algorithm 4: Bob $-$ RN_r hold segment

```
1 Bob generates $C'_{\text{RN}_r} = E_{PK_{\text{RN}_r}}(token, \nu, txid)$.
2 Bob looks up his routingTable and picks a tuple $(\text{RN}_r, \text{node}_k, p_k)$, with
    $p_k = (\text{hopCount}, \text{currMax}_s, \text{currMax}_r, t_e)$ where $\text{currMax}_r \geq \nu$ and sets
    $\text{hopMax} = \text{hopCount} + \text{pathStretch}$. Bob creates a tuple
    $(hold_r, \text{RN}_r, VK_{\text{RN}_r}, \nu, txid, C'_{\text{RN}_r}, \text{hopMax}, digest, te_1, te_2)$ and sends it to $\text{node}_k$.
3 for Each node ($\text{node}_i$) in the network do
4     Follow Algorithm 5
```

Determining te_1 and te_2 values: After Algorithm 2, RNs help senders determine te_1 and te_2 values for their transactions. A low value for te_1 and te_2 could result in premature timeout of a transaction when waiting a little longer would have resulted in the transaction completing successfully. te_1 and te_2 also shouldn't be so large that the liquidity in the network is locked up despite there being no viable paths via the involved RNs. The value of te_1 can be set by the sender based on a sampling of communication times

Algorithm 5: Subroutine for every node for *hold* and *pay* phase

```
Each node ($\text{node}_i$):
  Case 1: on receiving $hold_x$ message, $x \in \{s, r\}$, $msg = (hold_x, Y, VK_{\text{RN}_{(\cdot)}}, \nu, txid, C_{\text{RN}_{(\cdot)}},$
  $\text{hopMax}, digest, te_1, te_2)$, calls $hold(msg)$ defined in Algorithm 8.
  Case 2: on receiving $holdReject_x$ message $msg = (holdReject_x, Y, \text{VK}_{\text{RN}_{(\cdot)}}, \nu, txid)$ along
  with routingTable update, calls $holdReject(msg)$ defined in Algorithm 8.
  Case 3: on receiving $holdACK_x$ message $msg = (holdACK_x, t, \sigma_{\text{RN}_{(\cdot)}})$ along with
  routingTable update, calls $holdACK(msg)$ defined in Algorithm 8.
  Case 4: that did not receive a $holdACK_x$ tuple for a transaction $txid$, and current time $> te_1$,
  calls $holdACKTimeout()$ defined in Algorithm 8.
  Case 5: on receiving pay message $msg = (pay, preimage, \nu, txid)$, calls $pay(\text{msg})$ defined in
  Algorithm 8.
  Case 6: on receiving payACK message $msg = (payACK, \cdot, \cdot)$, calls $payACK(msg)$ defined in
  Algorithm 8.
  Case 7: that did not receive a payACK tuple for a transaction $txid$, and current time $> te_2$,
  calls $payACKTimeout()$ defined in Algorithm 8.
```

with its next-hop neighbors. For setting the value of te_2, each RN can estimate the communication time to its neighboring RNs, based on an estimate of number of hops per neighborhood and its estimated te_1; this can be built into the routing protocol with little overhead. This information can be broadcasted by RNs in their neighborhood (as part of the routing messages). When a sender sets the transaction's te_2, they can use the aggregate statistic of te_2 values they receive from their RN, e.g., 3 times the aggregate

te_2. We assume a certain amount of trial and error in finding the right multiplier on the part of the sender.

4.2 Hold phase

This is the first phase of transaction processing. In this phase, all nodes along a path from Alice to Bob will reserve or "hold" the amount Alice wishes to send to Bob. For ease of discussion, we divide the path into three segments, $\mathsf{Alice} - \mathrm{RN}_s$, $\mathrm{RN}_s - \mathrm{RN}_r$, and $\mathsf{Bob} - \mathrm{RN}_r$. Since the Alice-$\mathrm{RN}_s$ hold segment (Algorithm 3) and Bob-RN_r hold segment (Algorithm 4) are self-explanatory, due to space constraints, we describe them in Appendix 8.2.

Algorithm 6: RN operations in *hold* and *pay* phase.

```
1  if hold phase then
2      if $node_i == \mathrm{RN}_s$ then
3          $\mathrm{RN}_s$ on receiving ($hold_s, \mathrm{RN}_s, VK_{\mathrm{RN}_s}, \nu$, $txid$, $C_{\mathrm{RN}_s}$, hopMax,$digest$, $te_1$,$te_2$) tuple
             from a neighbor, does $m_{\mathrm{RN}_s} \leftarrow D_{SK_{\mathrm{RN}_s}}(C_{\mathrm{RN}_s})$ where
             $m_{\mathrm{RN}_s} = (VK_{\mathrm{RN}_r}, \nu, txid, C_{\mathrm{RN}_r})$.
4          $\mathrm{RN}_s$ looks up RNroutingTable to find a path ($\mathrm{RN}_k, \mathrm{RN}_{k+1}, \ldots \mathrm{RN}_l$) to $\mathrm{RN}_r$.
5          $\mathrm{RN}_s$ creates $m_{hold}$ = ($VK_{\mathrm{RN}_r}$, $\nu$, $txid$, $C_{\mathrm{RN}_r}$)
6          for $\mathrm{RN}_i$ in $\{\mathrm{RN}_l, \ldots, \mathrm{RN}_{k+1}, \mathrm{RN}_k\}$ do
7              $\mathrm{RN}_s$ does $m_{hold} = (PK_{\mathrm{RN}_i}, \nu, txid, E_{PK_{\mathrm{RN}_i}}(m_{hold}))$
8          $\mathrm{RN}_s$ then sends ($hold_s, Y, VK_{\mathrm{RN}_l}, \nu$, $txid$, $m_{hold}$, hopMax, $digest$, $te_1$,$te_2$), to its
             neighbor towards $Y$ according to RNroutingTable for selected path to $Y$ with
             $\mathrm{hopMax} = \mathrm{hopCount}$ of the path.
9          $\mathrm{RN}_s$ does $t = (txid, hold_s, \nu)$, $\sigma_{\mathrm{RN}_s} \leftarrow \mathsf{Sign}(\mathrm{sk}_{\mathrm{RN}_s}, t)$, sends ($holdACK_s$, $t$, $\sigma_{\mathrm{RN}_s}$)
             along with local routingTable to neighbor that sent $hold_s$.
10     else if $node_i == \mathrm{RN}_r$ then
11         if message is $hold_s$ then
12             When $\mathrm{RN}_r$ receives the $hold_s$ message, then the Alice $- \mathrm{RN}_r$ segment is complete.
                 $\mathrm{RN}_r$ does $t = (txid, hold_s, \nu)$, $\sigma_{\mathrm{RN}_r} \leftarrow \mathsf{Sign}(\mathrm{sk}_{\mathrm{RN}_r}, t)$, sends ($holdACK_s$, $t$,
                 $\sigma_{\mathrm{RN}_r}$) along with local routingTable to neighbor that sent $hold_s$.
13         else
14             When $\mathrm{RN}_r$ receives the $hold_r$ message, then the $\mathrm{RN}_r -$ Bob segment is complete.
                 $\mathrm{RN}_r$ sends $t = (txid, hold_r, \nu)$, $\sigma_{\mathrm{RN}_r} \leftarrow \mathsf{Sign}(\mathrm{sk}_{\mathrm{RN}_r}, t)$, sends
                 ($holdACK_r, t, \sigma_{\mathrm{RN}_r}$) along with local routingTable to neighbor that sent $hold_r$.
15     else if $\mathrm{node}_i = \mathrm{RN}_i, \forall \mathrm{RN}_i \in [\mathrm{RN}_k, \mathrm{RN}_{k+1}, \ldots \mathrm{RN}_l]$ then
16         $\mathrm{RN}_i$ on receiving the tuple ($hold_s, \mathrm{RN}_i, VK_{\mathrm{RN}_i}, \nu$, $txid$, $m_{hold}$, hopMax,$digest$,$te_1$,$te_2$)
             parses $m_{hold} = (PK_{\mathrm{RN}_i}, C_{\mathrm{RN}_i})$, sets
             $m_{hold} = (PK_{\mathrm{RN}_{i+1}}, C_{\mathrm{RN}_{i+1}}) \leftarrow D_{SK_{\mathrm{RN}_i}}(C_{\mathrm{RN}_i})$.
17         $\mathrm{RN}_i$ then sends ($hold_s, Y, VK_{\mathrm{RN}_{i+1}}, \nu$, $txid$, $m_{hold}$, hopMax, $digest$, $te_1$,$te_2$) to its
             neighbor towards $Y$ according to RNroutingTable for selected path to $Y$ with
             $\mathrm{hopMax} = \mathrm{hopCount}$ of the path.
18         $\mathrm{RN}_i$ does $t = (txid, hold_s, \nu)$, $\sigma_{\mathrm{RN}_i} \leftarrow \mathsf{Sign}(\mathrm{sk}_{\mathrm{RN}_i}, t)$, sends ($holdACK_s, t, \sigma_{\mathrm{RN}_i}$)
             along with local routingTable to neighbor that sent $hold_s$.
19 if pay phase then
20     $\mathrm{RN}_i$ on receiving pay tuple, sets $t$ = ($pay$, $txid$, $\mathrm{vk}_{\mathrm{RN}}$, $\nu$), does $\sigma_{\mathrm{RN}} \leftarrow \mathsf{Sign}(\mathrm{sk}_{\mathrm{RN}}, t)$.
21     $\mathrm{RN}_i$ then creates payACK tuple as ($payACK$, $t$, $\sigma_{\mathrm{RN}}$) to neighbor it had received pay tuple
         from.
```

Hold phase and Pay phase functions for intermediate nodes (Algorithm 5): This algorithm depicts the functions called by different nodes, i.e., regular/perimeter nodes and RNs, when they receive different messages during a SPRITE transaction (full details of the functions are in Appendix 8.2, Algorithm 8.) Let us now discuss when/why these functions are called by various nodes.

The *hold* function is called by a node on receiving a *hold*$_r$ or *hold*$_s$ message. The node checks its routing table and decides which neighbor the *hold* message needs to be forwarded to in order to route it to the target RN in the message. If no viable paths are available then the current node would forward a *holdReject* message to the neighbor from which it received the *hold* message originally. If a node in the network receives a *holdReject* message then it uses the *holdReject* function to process the message and make a decision about whether it should retry on other available paths or forward the *holdReject* message back in the direction of the sender.

holdACK and *payACK* functions are called by nodes in the network on receiving *holdACK* or *payACK* messages, respectively. These functions involve the verification of the received acknowledgment messages and forwarding them toward the sender on the transaction path. If a node in the network does not receive a *holdACK* or *payACK* message during the *hold* and *pay* phases, respectively, and the timers expire (te_1 for *hold* phase and te_2 for *pay* phase), then the respective nodes call the timeout functions, *holdACKTimeout* for *hold* phase and *payACKTimeout* for *pay* phase.

Hold phase and Pay phase RNs**' actions (Algorithm 6)**: We now discuss how the RNs handle operations in the hold phase, described in Algorithm 6. We recollect that RN_s is the first RN in the path, and RN_r is the last one. When RN_s receives a *hold*$_s$ message from Alice, it retrieves the verification key of RN_r (Line 3). RN_s then constructs an onion consisting of successive encryptions for all the RNs, $\{\mathrm{RN}_l, \ldots, \mathrm{RN}_k\}$ between RN_s and RN_r, with RN_r being the innermost layer of the onion. RN_s sends the onion to its next-hop neighbor along the path to RN_l (Line 3-8). Note that the intended recipient is the perimeter node common to RN_s and RN_l (since RN_s is not within hopMax distance of RN_l). RN_s also sends a signed *holdACK* message to Alice whom it received the *hold*$_s$ message from (Line 9). This is done to give the sender assurance that RN_s has received her message, but without requiring any blockchain writes. If malicious nodes drop *holdACK* messages, Alice will re-send the *hold*$_s$ tuple after a timeout. The honest intermediaries along the path will recognize the *hold*$_s$ message with the same $txid$ as a duplicate and will re-send the old, stored *holdACK* message along a different path. When RN_r receives the *hold*$_s$ tuple, she sends a *holdACK* tuple to Alice. Similarly, RN_r also sends a signed *holdACK* message back to Bob (Line 11-14). When an intermediate RN that is part of the onion created

Algorithm 7: Initialization of the *pay* phase.

```
1 At the end of hold phase (before $te_2$ expiry time) if $\mathrm{RN}_r$ has received a $hold_s$ tuple and a $hold_r$
     tuple with matching $token$, $\nu$, and $txid$ values, creates a message, $m = (proceedPay, txid, \nu)$,
     creates signature $\sigma_{proceedPay} = \mathsf{Sign}(SK_{\mathrm{RN}_r}, m)$ sends a tuple $(proceedPay, txid, \nu,$
     $\sigma_{proceedPay})$ towards Bob through Bob $- \mathrm{RN}_r$ segment.
2 On receiving the message Bob and Alice communicate out of band and Bob sends
     $(proceedPay, txid, \nu, \sigma_{proceedPay})$ tuple to Alice.
3 Bob creates a tuple $(pay, preimage, \nu, txid)$ and forwards it to its neighbor $node_o$ with $txid$
     towards $\mathrm{RN}_r$.
4 for Each node $node_i$ on $txid$ path on receiving pay message msg do
5     $\mathrm{node}_i$ calls $pay(\mathsf{msg})$ defined in Algorithm 8.
6 for Each node $node_i$ on $txid$ path on receiving payACK message msg do
7     $\mathrm{node}_i$ calls $payACK(msg)$ defined in Algorithm 8.
8 for Each node $node_i$ on $txid$ path that did not receive a payACK tuple and current time $> te_2$
     do
9     $\mathrm{node}_i$ calls $payACKTimeout()$ defined in Algorithm 8.
```

by RN_s receives *hold*$_s$, it peels off its layer, finds the identity of the next RN it needs to forward the message to, and sends the signed tuple to the perimeter node it knows can reach the destination RN (it finds this information from its *RNroutingTable*) (Line 15-18). Since PCNs are highly dynamic, there might be a situation during a transaction that an RN_i on the path between RN_s and RN_r cannot find a path to the next RN_{i+1}, even after the $\mathsf{maxRetries}$ number of retries. Neither the intermediate RNs nor any other non-RN nodes on the path can deviate from the original RN path defined by the onion created by RN_s. The intermediate nodes on each segment between two RNs do not know the next segment's target RN. In this case, the transaction needs to be failed all the way back to RN_s and then retried on a different path (different intermediate RNs) from RN_s to RN_r.

4.3 Pay phase

Algorithm 7: The *pay* phase is initialized by RN_r after it receives the *hold*$_s$ and *hold*$_r$ tuples originating from Alice and Bob respectively. Specifically, RN_r decrypts C_{RN_r} contained in *hold*$_s$ and C'_{RN_r} contained in *hold*$_r$, and compares the $token$ contained in both of them. If the $token$ is the same, that signifies to RN_r that some nodes Alice and Bob are sender and receiver in the transaction identified by $txid$, since only the two of them know $token$. RN_r then sends a signed *proceedPay* tuple to Bob, which signals the start of the pay phase. Bob forwards RN_r's *proceedPay* tuple to Alice to let her know the *pay* phase has started (Line 2). If RN_r does not receive $token$ in either *hold*$_s$ or *hold*$_r$, it sends a $\mathsf{multisig}(Rev, \cdot, \mathrm{RN}_r, \cdot, \cdot, \cdot, txid, \cdot)$ to its neighbor in the transaction path.

In the pay phase Bob's preceding neighbor along the path pays Bob first. Following this, each node pays its successor first, then gets paid back by its predecessor. Since nodes need some form of acknowledgment that the *pay* phase has gone through successfully, RN's that initiated the current segment send signed *payACK* tuples to the nodes in their segment (Algorithm 6, Lines 20, 21).

5 Security Analysis

We now discuss some potential attacks on SPRITE, and mitigation strategies, and then briefly discuss the formal analysis. We also give a phase-wise analysis of malicious activities in each of the bootstrapping, hold, and pay phases of SPRITE in Appendix 8.3.

5.1 Potential Attacks and Mitigation

Transaction malleability attack: A malicious RN_s colluding with a receiver Bob and RN_r might change the transaction amount ν to ν'. In the Alice-RN_s segment, the amount will be ν, the change occurs in the segments after that, all the way up until Bob.

Case 1: Let's assume $\nu' > \nu$ and $\delta = (\nu' - \nu)$. At the end of this attack, Alice has paid Bob ν coins and RN_s has paid Bob δ coins. None of the honest intermediaries will lose money: they get paid as many coins (by their successor) as they have paid along the path to their predecessor. The only entity losing money is RN_s since it will not get paid the δ amount and only get paid ν coins, tied to the tuple it received. *Case 2*: Let $\nu' < \nu$. If Bob, RN_s, and RN_r are all malicious, Bob will get paid ν' and RN_s will get paid the difference ($\delta = \nu - \nu'$) tied to the tuple received from Alice with ν coins, but since they were both collaborating malicious entities, this does not affect honest intermediaries. If Bob is honest, then Bob will get paid ν, but RN_s will send a lower amount ν' to RN_r, thus making malicious RN_r lose money. In all the above cases the adversaries end up losing money but none of the honest nodes get less coins than what they paid, hence we do not consider these to be successful attacks on SPRITE.

Transaction forgery attack: We assume no honest users in the system will share their signing keys related to SPRITE with other users. This avoids any situations where an adversary can communicate on a channel created between two neighbors on behalf of one of them (e.g., Alice/Bob $\rightarrow$ Craig, where Alice and Bob share a channel and Bob is malicious), or the adversary can sign contracts on behalf of an honest Alice without Alice's knowledge (e.g., Bob $\rightarrow$ Alice $\rightarrow$ Craig, where Bob is malicious). If any user's keys are leaked then that user will generate a new set of keys and notify all her neighbors about the new keys. One could use forward-secure signatures [15] for invalidating the old leaked keys.

Sybil/Counting-based attack: An adversary could intercept network communications over time, isolating *hold*$_x$ messages, and associating messages sharing the same $txid$ and *digest*. The adversary will try to identify the sender/receiver in a transaction by isolating messages with the highest hopMax or lowest timer values.

Counting number of hops based on hopMax does not reveal the identity of sender/receiver since each RN resets the hopMax value for each segment. The hopMax value is decremented by each node and is an estimate of the expected $\mathrm{hopCount}$ to the target routing helper in the current segment, and tells how far the current message should go before being dropped. This does not leak to a node in the network information about how far the sender of the current received message was from it (intermediate nodes do not know which segment they are a part of). Since te_1 and te_2 are system parameters and are included in the hold messages, all nodes in the network will receive the same value of te_1 and te_2. On receiving the hold message, each node locally computes its timeout values $te_1.txid$ and $te_2.txid$, and does not forward the local values further.

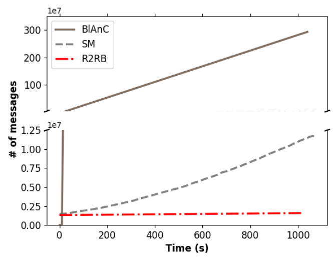
Sender refusing to pay: Whenever there are timeouts in the hold phase for a specific segment, the sender RN for that segment will retry the hold phase on a different path. If there are timeouts in the pay phase the nodes that timed out in that specific segment, will publish their hold and pay contracts on a public repository or blockchain. Since the hold and pay contracts are signed with pseudonymous identities, this does not leak information about nodes to the public, but neighbors know each others' identities and if a node does not post a pay contract associated with a hold contract then this identifies the malicious activity to the whole network. Any honest neighbors will then avoid the malicious node for subsequent retries and transactions. If the sender is the malicious node, then all nodes on the path need to discard the hold and pay contracts and roll back the transaction since the sender has been identified as malicious and the sender-RN_s segment will not be retried.

5.2 Formal Security Analysis

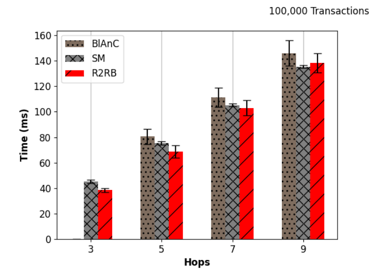
We analyze the security of SPRITE in the Universal Composability framework [16]. To this end, we define an ideal functionality, $\mathcal{F}_{\mathsf{SPRITE}}$, consisting of three functionalities, $\mathcal{F}_{\text{setup}}$, $\mathcal{F}_{\text{hold}}$, and $\mathcal{F}_{pay}$. We use two helper functionalities from [16], $\mathcal{F}_{\mathsf{sig}}$ and $\mathcal{F}_{\mathsf{smt}}$, to model ideal functionalities for digital signatures and secure/authenticated channels, respectively. We assume the ideal functionalities share internal state and can access data stored internally by each other.

$\mathcal{F}_{\text{setup}}$ models the broadcast phase where nodes register and establish payment channels and RNs register and make known their verification key to other nodes in the network. It also provides functionality for broadcasting messages such as *RN-Update* and *RN-UpdateReply*. $\mathcal{F}_{\text{hold}}$ provides interfaces for creating a *hold*$_s$ message from sender and *hold*$_r$ message from receiver, RN-specific *hold* phase functionalities, and the pairwise contract multisig functionality. $\mathcal{F}_{\text{pay}}$ provides interfaces specific to the *pay* phase, creation and verification of a *pay* message, pairwise contracts creation and signing in *pay* phase, etc. We assume that all functionalities in $\mathcal{F}_{\text{sprite}}$ have access to a global clock from which they can obtain the current time. We give the proof of the following theorem along with the functionalities in Appendix 8.4.

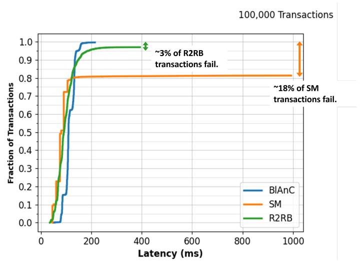
Theorem 5.1. *Let $\mathcal{F}_{sprite}$ be an ideal functionality for SPRITE. Let $\mathcal{A}$ be a probabilistic polynomial-time (PPT) adversary for SPRITE, and let $\mathcal{S}$ be an ideal-world PPT simulator for $\mathcal{F}_{sprite}$. SPRITE UC-realizes $\mathcal{F}_{sprite}$ for any PPT distinguishing environment $\mathcal{Z}$.*



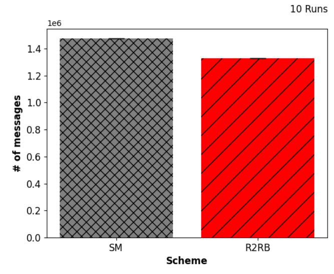
(a) # of messages (Y-Axis is split to account for messages' explosion in BlAnC).



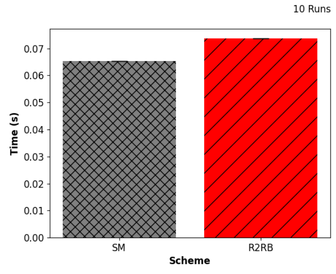
(b) Average latencies with respect to transaction hop-count.



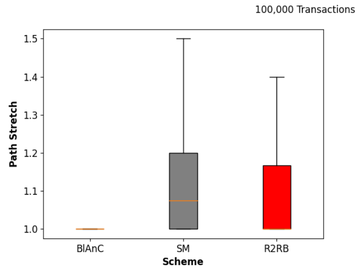
(c) CDF of Transaction latencies.



(d) # of messages for the Bootstrap phase.



(e) Duration of the Bootstrap phase.



(f) Path-stretch of transactions.

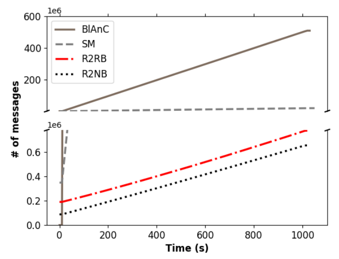
Figure 3: Results for simulations in the Lightning Topology (LT).

6 Experimental Analysis

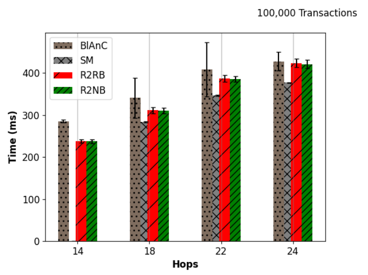
6.1 Experimental Setup

We compared R2RB and R2NB with BlAnC [41] and Speedy Murmurs [49] (referred to as SM in this section), across two topology types, ten topologies each. The first topology, referred to as LT, was taken from the publicly available Lightning gossip dataset [2] from May 31, 2022. The network has 15833 nodes and 156072 channels. We removed any nodes that did not have any outgoing connections along with 80% of the nodes which had one incoming or outgoing connection (these nodes are not involved in routing), leaving 8995 nodes and 129724 channels in LT. We designated the top 10 highly connected nodes as RNs for evaluating R2RB and BlAnC, and to act as landmarks in SM. As channel capacity is not present in the gossip messages from the Lightning data, we choose the maximum allowed amount for a single transaction as the link weight as this value should correlate to a realistic channel capacity. We compare the performance of BlAnC, SM, and R2RB on LT. R2NB is not applicable to LT due to the closely located RN nodes.

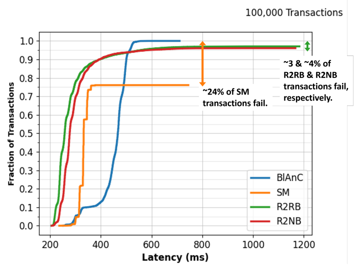
We constructed a second privacy-preserving network topology (PPNT) as described in Section 4, to evaluate R2RB, R2NB, BlAnC, and SM. We start by taking the RNs in LT and start adding nodes to the PCN where the initial few nodes set up payment channels with RNs but subsequent nodes joining the network connect to other regular nodes, thus forming layers around the RNs. We add nodes until each RN has a diameter of about 7 hops and a neighborhood of roughly 800 nodes. The perimeter nodes of each neighborhood are randomly connected to perimeter nodes belonging to other RN neighborhoods. PPNT had 7978 nodes and 25302 channels. The link weights used in this topology are similar to LT. We categorized the link weights from LT into two groups, the first group contained channels with at least one highly connected node (RN), and the second group was made up of links between two regular nodes. The link weights were then randomly sampled from these two groups and assigned to the links



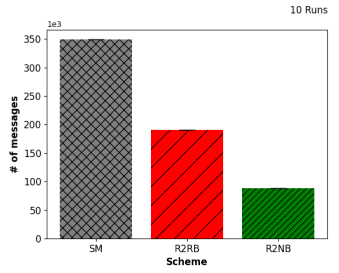
(a) # of messages (Y-Axis is split to account for messages' explosion in BlAnC).



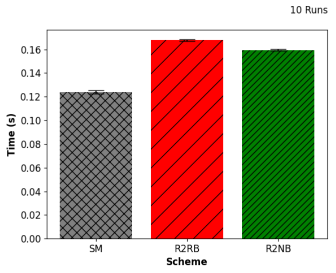
(b) Average latencies with respect to transaction hop-count.



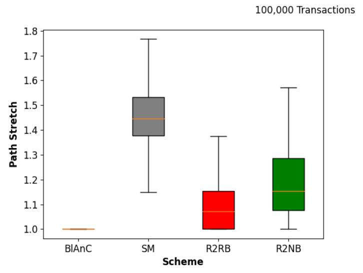
(c) CDF of Transaction latencies.



(d) # of messages for the Bootstrap phase.



(e) Duration of the Bootstrap phase.



(f) Path stretch of transactions.

Figure 4: Results for simulations in the Privacy Preserving Network Topology (PPNT).

in PPNT based on the channel type.

We randomly chose senders and receivers with at least 3 and 8 hops between them for LT & PPNT respectively. Although publicly available data for the Lightning network claims an average of $22k$ transactions per day [1] – significantly lower than 10 transactions/second, we set a transaction rate at 10 transaction/second. This high rate was used to assess the scalability of SPRITE. In SM, each transaction gets split into 10 uniform sub-transactions, one for each RN (referred to as landmarks in SM).

We implemented R2RB, R2NB, SM and BlAnC, and deployed the generated topologies in the ns-3 simulator [3] for our experiments.[3] The results were averaged over 10 runs with PPNT for a total of $100k$ transactions. The simulations were run on a Desktop class machine with Intel(R) Core(TM) i7-10700 @ 3.8 GHz CPU and 64 GB of RAM. The metrics for comparison are: path stretch (ratio of the hop-count of a completed transaction to the optimal hop-count), end-to-end transaction processing time (latency), transaction success rate, set up costs during Bootstrap phase (message complexity and duration), and the overall message complexity of the entire simulation.

6.2 Experimental Results

LT Topology results: Figure 3a shows the growth of the message complexity within LT over time. BlAnC inundates the network with broadcasts for each transaction and given the interconnected nature of LT this results in a dramatic increase in message complexity, growing at a rate roughly 100 times that of SM. SM, while having only a fraction of the number of messages compared to BlAnC, still grows at a much faster rate than R2RB. This is attributable to the splitting of each transaction and the acknowledgments sent back on the payment path in the routing phase.

Figure 3b shows the growth of latency with respect to hop-count. BlAnC has a higher latency compared to R2RB and SM. This is attributed to BlAnC having three phases as opposed to two in

[3]Code: https://github.com/nsol-nmsu/sprite

SM and R2RB. Note that given its sub-optimality, BlAnC never chooses a 3-hop sender-receiver path. SM and R2RB have similar latencies. We model cryptographic operations for both R2RB and BlAnC, but not for SM. We also do not model the delay imposed by blockchain operations for BlAnC. The hop-counts of transactions in LT range between 3-10 hops, with BlAnC, SM, and R2RB having average hop-counts of 7, 5, and 6, respectively. The hop-counts for SM represent the highest across the hop-counts of all the split transactions.

Figure 3c shows the cumulative distribution function (CDF) of latencies for all transactions. Both R2RB and SM outperform BlAnC significantly, which had an average latency of 113.5 ms, while SM and R2RB had average latencies of 79.6 ms and 95.3 ms, respectively. The additional delay in BlAnC is on account of the extra broadcast-based $Find$ phase. R2RB is able to perform almost as well as SM in terms of real-world delays while providing significantly more security and privacy guarantees. It also has a significantly higher transaction success rate at 97.17% compared to SM's 81.3%. R2RB outperforms SM in terms of success rate due to our in-network retry mechanism, as well as *routingTable* updates that are propagated within the network for each *holdACK* and *holdReject* message. In LT 9.864% of transactions required a retry attempt for R2RB. Due to the design of BlAnC, the sender can only send the maximum available credits on the fastest path to the receiver, hence, only 69.06% of transactions sent the full amount of required credits. For practical applications, these transactions can be repeated by splitting the larger ones into sub-transactions, similar to SM.

Figure 3d shows the total number of messages required to bootstrap the network with routing information while Figure 3e shows the duration of the phase. BlAnC is excluded from this comparison as it does not have a Bootstrap phase. SM requires more messages for its bootstrapping phase in LT than R2RB but takes about 10 ms less than R2RB to complete this phase.

The path stretch of transactions is shown in Figure 3f, it should be noted that BlAnC always finds the most optimal path in terms of hop-count due to its broadcast-based pathfinding mechanism. The path stretch for SM was calculated by taking the average amount of hops taken by each sub-transaction and comparing that against the optimal path (obtained from Dijkstra's algorithm) between the sender and the receiver. For R2RB and BlAnC, the number of hops taken by a transaction were compared against the total hops in the corresponding optimal paths between the sender & RN_s, RN_s & RN_r, and RN_r & the receiver. SM incurs the worst path stretch with a median of 1.075, while R2RB has a median path stretch of 1.0. The variation in path stretch for transactions in R2RB is due to the *routingTable* of nodes becoming stale as the simulation progresses with new transactions. The routing tables can remain fresh by issuing periodic broadcasts from RNs, similar to the Bootstrap phase, to update the *routingTable* of nodes. The higher path stretch in SM can be attributed to its embedded prefix routing and splitting of transactions among different paths.

PPNT Topology results: All four schemes show linear growth in the number of messages as seen in Figure 4a. With R2RB and R2NB the number of messages is the lowest and continues to grow linearly at these low values. As with LT, BlAnC's $Find$ phase results in thousand times more messages than R2RB and R2NB while SM results in a ten times higher number of messages in comparison.

Figure 4b shows the growth of latency with respect to hop-count. Both R2RB and R2NB have a slightly larger latency for each transaction of a given hop-count when compared to SM due to the cryptographic operations between pairs of nodes on the path.

Figure 4c shows a CDF where it can be observed that for the majority of transactions, R2RB and R2NB have lower latencies than BlAnC and SM while maintaining transaction success rates of 97.01% and 96.02%. SM on the other hand, has a success rate of 76.08%. Roughly five percent of transactions in SPRITE (R2RB and R2NB) have higher latencies than those found in BlAnC and SM due to SPRITE's in-network retries that would otherwise fail.

The number of messages and the duration of the Bootstrap phase were averaged over ten runs; results shown in Figure 4d and 4e. In contrast to LT, SM has higher number of messages, with an average of around $348k$ messages when compared to R2RB with $190k$ messages and R2NB with $88k$

messages respectively. The higher complexity of R2RB and SM is due to the more distributed nature of the PPNT network, where the landmarks have a much lower degree than LT. In the case of SM, this results in more nodes receiving multiple messages for each landmark advertisement compared to LT. Transactions in SM take the least amount of time, while R2RB takes the most, but similar to LT the difference is negligible.

The transaction path stretch in Figure 4f shows that BlAnC is the most efficient in terms transaction path length. This is because it finds the most optimal path in terms of hop-count due to its broadcast-based pathfinding mechanism. This optimal path stretch does come at the cost of higher overhead and much higher latencies.

The median path stretch value for SM is 1.45 and is significantly higher than R2RB and R2NB with median values of 1.07 and 1.15, respectively. Due to the distributed network topology, the prefix-based embedding system used by SM does not adequately identify the shortest path when landmarks are far from either the sender or receiver. R2NB's inefficiency is due to the unknown distance of the chosen perimeter node from the next RN.

7 Conclusion

In this paper, we present SPRITE, a secure, privacy-preserving, and efficient routing protocol for payment channel networks. SPRITE can support concurrent transactions and takes two rounds of communication for pathfinding and routing transactions, which is optimal. There are several interesting directions for future work. One is investigating the design of economic models and mechanisms for estimating and optimizing routing fees for both, regular nodes and routing nodes in a PCN. Another direction of future work is studying the interplay of PCNs with other Layer-2 solutions. Finally, another interesting direction for future work is to mechanically verify the proof of security of SPRITE (and potentially other PCN protocols) using interactive theorem provers such as EasyUC [17, 14]. This would be a valuable contribution to the PCN literature, since, to the best of our knowledge, UC proofs in PCNs till date have not used interactive theorem provers.

References

[1] The growth of the lightning network. `https://k33.com/research/archive/articles/the-growth-of-the-lightning-network`.

[2] Lightning network gossip datasets and topology. `https://github.com/lnresearch/topology`.

[3] ns-3 network simulator. `https://www.nsnam.org/`.

[4] Ripple data. `https://data.ripple.com/`.

[5] Visa fact sheet. `https://www.visa.co.uk/dam/VCOM/download/corporate/media/visanet-technology/aboutvisafactsheet.pdf`.

[6] What is the lightning network in bitcoin and how does it work? `https://cointelegraph.com/bitcoin-for-beginners/what-is-the-lightning-network-in-bitcoin-and-how-does-it-work`.

[7] Xrpscan. `https://xrpscan.com/`.

[8] Phoenix wallet 4: Trampoline payments. `https://medium.com/ACINQ/phoenix-wallet-part-4-trampoline-payments-fb1befd027c8`, 2023. Accessed: 2023-12-19.

[9] Trampoline routing. `https://lists.linuxfoundation.org/pipermail/lightning-dev/2019-August/002100.html`, 2023. Accessed: 2023-12-19.

[10] What are trampoline payments. `https://thebitcoinmanual.com/articles/btc-trampoline-payments/`, 2023. Accessed: 2023-12-19.

[11] Abdelrahaman Aly, Edouard Cuvelier, Sophie Mawet, Olivier Pereira, and Mathieu Van Vyve. Securely solving simple combinatorial graph problems. In Ahmad-Reza Sadeghi, editor, *Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*, volume 7859 of *Lecture Notes in Computer Science*, pages 239–257. Springer, 2013.

[12] Abdelrahaman Aly and Mathieu Van Vyve. Securely solving classical network flow problems. In Jooyoung Lee and Jongsung Kim, editors, *Information Security and Cryptology - ICISC 2014 - 17th International Conference, Seoul, Korea, December 3-5, 2014, Revised Selected Papers*, volume 8949 of *Lecture Notes in Computer Science*, pages 205–221. Springer, 2014.

[13] Andreas Anotonopoulos, Olaoluwa Osuntokun, and Rene Pickhardt. Mastering the lightning network. `https://github.com/lnbook/lnbook`.

[14] Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, and Pierre-Yves Strub. Mechanized proofs of adversarial complexity and application to universal composability. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 2541–2563. ACM, 2021.

[15] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 431–448. Springer, 1999.

[16] Ran Canetti. Universally composable signature, certification, and authentication. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.*, 2004.

[17] Ran Canetti, Alley Stoughton, and Mayank Varia. Easyuc: Using easycrypt to mechanize proofs of universally composable security. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, pages 167–183. IEEE, 2019.

[18] Yanjiao Chen, Yuyang Ran, Jingyue Zhou, Jian Zhang, and Xueluan Gong. Mpcn-rp: A routing protocol for blockchain-based multi-charge payment channel networks. *IEEE Transactions on Network and Service Management*, 19(2):1229–1242, 2022.

[19] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed E. Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains - (A position paper). In Jeremy Clark, Sarah Meiklejohn, Peter Y. A. Ryan, Dan S. Wallach, Michael Brenner, and Kurt Rohloff, editors, *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*, volume 9604 of *Lecture Notes in Computer Science*, pages 106–125. Springer, 2016.

[20] Christian Decker. Lightning network research - topology datasets. `https://github.com/lnresearch/topology`.

[21] Lisa Eckey, Sebastian Faust, Kristina Hostáková, and Stefanie Roos. Splitting payments locally while routing interdimensionally. *IACR Cryptol. ePrint Arch.*, 2020:555, 2020.

[22] Felix Engelmann, Henning Kopp, Frank Kargl, Florian Glaser, and Christof Weinhardt. Towards an economic analysis of routing in payment channel networks. In *Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, SERIAL '17, pages 2:1–2:6, 2017.

[23] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert van Renesse. Bitcoin-ng: A scalable blockchain protocol. In Katerina J. Argyraki and Rebecca Isaacs, editors, *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, pages 45–59. USENIX Association, 2016.

[24] L.R. Ford and D.R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8, 1954.

[25] A.V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. *J. of ACM*, 35:921–940, 1988.

[26] Qianyun Gong, Chengjin Zhou, Le Qi, Jianbin Li, Jianzhong Zhang, and Jingdong Xu. VEIN: high scalability routing algorithm for blockchain-based payment channel networks. In *20th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2021, Shenyang, China, October 20-22, 2021*, pages 43–50. IEEE, 2021.

[27] Hsiang-Jen Hong, Sang-Yoon Chang, and Xiaobo Zhou. Auto-tune: Efficient autonomous routing for payment channel networks. In *2022 IEEE 47th Conference on Local Computer Networks (LCN)*, pages 347–350, 2022.

[28] Heba Kadry and Yasser Gadallah. A machine learning-based routing technique for off-chain transactions in payment channel networks. In *2021 IEEE International Conference on Smart Internet of Things (SmartIoT)*, pages 66–73, 2021.

[29] Heba Kadry and Yasser Gadallah. A machine learning-based routing technique for off-chain transactions in payment channel networks. In *IEEE International Conference on Smart Internet of Things, SmartIoT 2021, Jeju, Republic of Korea, August 13-15, 2021*, pages 66–73. IEEE, 2021.

[30] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 357–388. Springer, 2017.

[31] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 583–598. IEEE Computer Society, 2018.

[32] MIT Media lab. Digital currency initiative. Layer 2: The lightning network. `https://dci.mit.edu/lightning-network`.

[33] Lightning network. `https://lightning.network/`.

[34] Lightning network routing nodes. `https://docs.lightning.engineering/the-lightning-network/multihop-payments/what-makes-a-good-routing-node`.

[35] Siyi Lin, Jingjing Zhang, and Weigang Wu. FSTR: funds skewness aware transaction routing for payment channel networks. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*, pages 464–475. IEEE, 2020.

[36] G. Malavolta, P. Moreno-Sanchez, A. Kate, and M. Maffei. Silentwhispers: Enforcing security and privacy in decentralized credit networks. In *Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[37] G. Malavolta, P. Moreno-Sanchez, A. Kate, M. Maffei, and S. Ravi. Concurrency and privacy with payment-channel networks. In *Proceedings ACM SIGSAC Conference on Computer and Communications Security, CCS*, pages 455–471, 2017.

[38] Andrew Miller, Iddo Bentov, Surya Bakshi, Ranjit Kumaresan, and Patrick McCorry. Sprites and state channels: Payment networks that go faster than lightning. In *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*, pages 508–526, 2019.

[39] Andrew Miller, Ari Juels, Elaine Shi, Bryan Parno, and Jonathan Katz. Permacoin: Repurposing bitcoin work for data preservation. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 475–490. IEEE Computer Society, 2014.

[40] P. Moreno-Sanchez, A. Kate, M. Maffei, and K. Pecina. Privacy preserving payments in credit networks: Enabling trust with privacy in online marketplaces. In *Annual Network and Distributed System Security Symposium, NDSS*, 2015.

[41] Gaurav Panwar, Satyajayant Misra, and Roopa Vishwanathan. Blanc: Blockchain-based anonymous and decentralized credit networks. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, CODASPY*, pages 339–350, 2019.

[42] Sunoo Park, Albert Kwon, Georg Fuchsbauer, Peter Gazi, Joël Alwen, and Krzysztof Pietrzak. Spacemint: A cryptocurrency based on proofs of space. In Sarah Meiklejohn and Kazue Sako, editors, *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers*, volume 10957 of *Lecture Notes in Computer Science*, pages 480–499. Springer, 2018.

[43] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Spot-light: Lightweight private set intersection from sparse OT extension. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part III*, volume 11694 of *Lecture Notes in Computer Science*, pages 401–431. Springer, 2019.

[44] David Pointcheval and Olivier Sanders. Short randomizable signatures. In Kazue Sako, editor, *Topics in Cryptology - CT-RSA 2016 - The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29 - March 4, 2016, Proceedings*, volume 9610 of *Lecture Notes in Computer Science*, pages 111–126. Springer, 2016.

[45] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. `https://lightning.network/lightning-network-paper.pdf`.

[46] Raiden network. `https://raiden.network/`.

[47] Ripple. `https://ripple.com`.

[48] S. Roos, P. Moreno-Sanchez, A. Kate, and I. Goldberg. Settling payments fast and private: efficient decentralized routing for path-based transactions. In *Annual Network and Distributed System Security Symposium, NDSS*, 2018.

[49] 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.

[50] Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Kathleen Ruan, Parimarjan Negi, Lei Yang, Radhika Mittal, Giulia C. 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.

[51] B. Viswanath, M. Mondal, P. K. Gummadi, A. Mislove, and A. Post. Canal: scaling social network-based sybil tolerance schemes. In *Proceedings of EuroSys*, pages 309–322, 2012.

[52] Peng Wang, Hong Xu, Xin Jin, and Tao Wang. Flash: Efficient dynamic routing for offchain networks. CoNEXT '19, page 370–381, New York, NY, USA, 2019. Association for Computing Machinery.

[53] Ruozhou Yu, Guoliang Xue, Vishnu Teja Kilari, Dejun Yang, and Jian Tang. Coinexpress: A fast payment routing mechanism in blockchain-based payment channel networks. In *27th International Conference on Computer Communication and Networks, ICCCN 2018, Hangzhou, China, July 30 - August 2, 2018*, pages 1–9. IEEE, 2018.

[54] Xiaoxue Zhang, Shouqian Shi, and Chen Qian. Webflow: Scalable and decentralized routing for payment channel networks with high resource utilization. *CoRR*, abs/2109.11665, 2021.

[55] Yuhui Zhang and Dejun Yang. Robustpay: Robust payment routing protocol in blockchain-based payment channel networks. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, pages 1–4, 2019.

[56] Yuhui Zhang and Dejun Yang. Robustpay$^+$: Robust payment routing with approximation guarantee in blockchain-based payment channel networks. *IEEE/ACM Trans. Netw.*, 29(4):1676–1686, 2021.

[57] Yuhui Zhang, Dejun Yang, and Guoliang Xue. Cheapay: An optimal algorithm for fee minimization in blockchain-based payment channel networks. In *2019 IEEE International Conference on Communications, ICC 2019, Shanghai, China, May 20-24, 2019*, pages 1–6. IEEE, 2019.

8 Appendix

8.1 AS Function Definitions

Definition 8.1. *(Sequential Aggregate Signatures [44]). Let* $\mathbb{G}_1$, $\mathbb{G}_2$ *be prime-order cyclic groups of size* p*, such that* $g \in \mathbb{G}_1$, $\widetilde{g} \in \mathbb{G}_2$*, and* $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$.

- $\mathsf{AS.Setup}(1^k)$*: Given a security parameter* k*, this algorithm selects a random* $x \in \mathbb{Z}_p$ *and outputs* $pp \leftarrow (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g, X, \widetilde{g}, \widetilde{X})$*, where* $X = g^x$ *and* $\widetilde{X} = \widetilde{g}^x$.
- $\mathsf{AS.KeyGen}(pp)$*: This algorithm selects a random* $y \leftarrow\!\$\ \mathbb{Z}_p$*, computes* $\widetilde{Y} \leftarrow \widetilde{g}^y$ *and sets* sk *as* y *and* pk *as* Y.

- $\mathsf{AS.Sign}(\mathsf{sk}, \sigma, (m_1, \ldots, m_r), (\mathsf{pk}_1, \ldots, \mathsf{pk}_r), m)$ *proceeds as follows:*
  - *If* $r = 0$*, then* $\sigma \leftarrow (g, X)$*;*
  - *If* $r > 0$ *but* $\mathsf{AS.Verify}(\ (\mathsf{pk}_1, \ldots, \mathsf{pk}_r), \sigma, (m_1, \ldots, m_r)) = 0$*, then it halts;*
  - *If* $m = 0$*, then it halts;*
  - *If for some* $j \in \{1, ..., r\}$ $\mathsf{pk}_j = \mathsf{pk}$*, then it halts.*

  If the algorithm did not halt, then it parses sk *as* y *and* σ *as* (σ_1, σ_2)*, selects* $t \leftarrow\!\$\ \mathbb{Z}_p$ *and computes* $\sigma' = (\sigma'_1, \sigma'_2) \leftarrow (\sigma_1^t, (\sigma_2 \cdot \sigma_1^{y \cdot m})^t)$*. It eventually outputs* σ'*.*

- $\mathsf{AS.Verify}((\mathsf{pk}_1, \ldots, \mathsf{pk}_r), (m_1, \ldots, m_r), \sigma)$ *parses* σ *as* (σ_1, σ_2) *and* pk_j *as* $\widetilde{Y}_j$*, for* $j = 1, \ldots, r$*, and checks whether* $\sigma_1 \neq 1_{\mathbb{G}_1}$ *and* $e(\sigma_1, \widetilde{X} \cdot \prod \widetilde{Y}_j^{m_j} = e(\sigma_2, \widetilde{g}$ *are both satisfied. In the positive case, it outputs* 1*, and* 0 *otherwise.*

8.2 Algorithms

$\mathsf{Alice} - \mathrm{RN}_s$ **segment (Algorithm 3)**: Alice first picks an RN_s that she can find a short path to via her neighbors in her *routingTable*, and Bob picks RN_r similarly. Bob then picks a *preimage* whose hash digest is sent out-of-band to Alice (Line 1). Alice creates two ciphertexts for RN_s and RN_r, encrypted using $\mathrm{VK}_{\mathrm{RN}_s}$ and $\mathrm{VK}_{\mathrm{RN}_r}$ that contain a $token$, a unique $txid$ and amount ν she wants to send (Line 3). The digest and $token$ will be used later in the *pay* phase. She includes $\mathrm{VK}_{\mathrm{RN}_r}$ inside RN_s's ciphertext to ensure RN_s knows which RN to route to. Alice refers to her *routingTable* and finds a neighbor with the shortest path to RN_s that has sufficient liquidity, and sends the *hold*$_s$ tuple to that neighbor. Due to network dynamics, the $\mathrm{hopCounts}$ stored in her *routingTable* could have become stale, which is where $\mathrm{pathStretch}$ helps. When a *hold*$_s$ travels from Alice to RN_s, every node will try for $\mathrm{pathStretch}$ number of hops more than the hopMax that Alice stipulated in the *hold*$_s$ tuple.
$\mathsf{Bob} - \mathrm{RN}_r$ **segment (Algorithm 4)**: This algorithm depicts the hold phase on the $\mathsf{Bob} - \mathrm{RN}_r$ segment. Bob refers to his *routingTable* to find the RN that is reachable from any of his neighbors with minimum hops, RN_r. Bob then generates a ciphertext C'_{RN_r} containing the unique $txid$, amount ν, and unique $token$ transmitted to him out of band by Alice in Algorithm 3. The idea here is that RN_r upon receiving and decrypting two copies of the $token$ independently from Alice and Bob will be able to connect them as sender and receiver in the transaction identified by $txid$, which signifies the hold phase has concluded successfully. The $token$ is known only to Alice and Bob, unlike other transaction parameters such as $txid$, which are known by (potentially malicious) intermediaries in plaintext.

Subroutine for intermediate node (Algorithm 8): This algorithm details the functions called by nodes in the network during *hold* and *pay* phases discussed in Algorithm 5. *hold* function is called by different nodes when they receive a *hold*$_s$ or *hold*$_r$ tuple. If the receiving node happens to be a perimeter node, it checks if the tuple was addressed to it (line 5). If so, and if it has a path to transmit the asking amount, it updates the *hold*$_s$ tuple with the destination RN as the receiver and forwards the message. Nodes in the network that have a path to the target RN or perimeter node, set the transaction's local retry counter to be zero. They also store the transaction digest and identity of the destination RN, initialize the timers te_1 and te_2, forward the message to their neighbor, and sign a $\mathsf{multisig}$ *hold* contract with their neighbor. Each node also sets its transaction-specific timers, $te_1.txid$ and $te_2.txid$. If the node is the target RN node in the *hold* message, it follows Algorithm 6 to process the message.

If the node is a non-perimeter, non-RN node, it signs a $\mathsf{multisig}$ contract with its neighbor and forwards the message to its next-hop neighbor after checking if it has a viable path with sufficient liquidity to the destination RN/perimeter node (lines 11-16). Let us now focus on nodes setting their timers. Each Node computes two transaction-specific timers $txid.te_1$ and $txid.te_2$ as functions of the current time $\mathrm{currTime}$ and the global te_1 and te_2 (line 13, 15). We do this since each node that is not a destination (either perimeter node or an RN) cannot simply use te_1 and te_2 for timeouts, since we need to ensure that timeouts of nodes are staggered, i.e., the node closest to a destination times out first, whereas the nodes farthest away from the destination time out last. If a node receives multiple copies of a *hold*$_x$ message (where $x \in \{s, r\}$ corresponding to sender and receiver respectively) associated with the same $txid$, it ignores subsequent messages. When a node cannot find any viable path to a destination (even after retries), it sends a *holdReject* message back to its predecessor from whom it received the *hold*$_x$ message. It also sends a copy of its current *routingTable* so the predecessor can update its own *routingTable*. The multisig contract that was signed associated with the current transaction is also deleted by both nodes (lines 1-4).

The *holdReject* function is called by a node in the network on receiving a *holdReject* message. Nodes in the network could choose to retry on other paths upto maxRetries limit (lines 18-19) or choose to forward *holdReject* message back towards the sender if path is unavailable to a target RN (line 23). If Alice or Bob receive a *holdReject*, that means that there is no viable path from their neighbor to RN_s (for Alice) or RN_r (for Bob), hence they need to retry the transaction with different neighbors and possibly new RNs (line 21). The way RNs handle *hold*$_x$, *holdReject* messages is slightly different; we discuss RNs' actions in these phases separately in Algorithm 6. The *holdACK* function is called by a node on receiving a *holdACK*$_x$ message. When the target RN in a given segment receives a *hold*$_x$ tuple, and is willing to process the transaction, it sends back a *holdACK* tuple containing its signature to the neighboring node it received the *hold*$_x$ tuple from.

When a node receives a *holdACK*$_x$ tuple, it deletes its timer te_1, and forwards the *holdACK*$_x$ to its predecessor. Ultimately, the sender should receive the *holdACK*$_x$ tuple, which will tell her that the *hold*$_x$ messages along that segment were successful, and reached the destination RN. If a node does not receive a *holdACK* tuple until te_1 expires, it calls the *holdACKTimeout* function, which retries the transaction if possible, else it drops the transaction and sends a *holdReject*$_x$ tuple towards the sender (line 28). If any node including an RN receives a *hold* tuple on a different path, after it has already forwarded a *holdACK* towards Alice on another path, then it should replay the *holdACK* tuple on the new path and send a *holdReject* message on the previous path. This accounts for a malicious node on any path not forwarding *holdACK* tuples downstream thus timing out nodes preceding it, and prompting the formation of another path to the target RN.

The *pay* function is called by nodes in the network on receiving a *pay* tuple from the direction of the receiver. The pay function involves verification of information included in the *pay* tuple like the *digest* to guard against malicious receivers in the network (line 29) before it is forwarded towards the sender along with updating the link weights with the neighbor towards the sender. The *payACK* function is used to confirm that the *pay* message has reached the RN towards the sender on the current segment. The *payACK* tuple helps verify that the acknowledgment has been sent by an authorized RN or by the sender Alice for the first segment. On successful verification of the acknowledgment, the node clears the timer te_2 associated with the *pay* phase before forwarding to their predecessor along the path (Lines 35-38). Finally, there is a subtle issue we need to deal with: all RNs can send signed *payACK* messages to their segment, since nodes need to know whether to time out and cancel the multisig contracts, or whether the transaction went through successfully. RNs can sign tuples since their verification keys are known to all users in the PCN. In the Alice $-$ RN_s segment though, since Alice's identity is not publicly known, she cannot sign and send a *payACK* tuple. Yet nodes on the Alice $-$ RN_s segment still need to know whether the transaction has gone through or whether they need to revoke their multisig contracts. We address this issue by having Alice reveal the preimage of the $txid$, *preimage*$_{txid}$ to all nodes in the Alice $-$ RN_s segment (Line 33-34). Nodes on Alice $-$ RN_s segment then either clear timer te_2, or revoke their multisig contracts as appropriate (Lines 39-41). RN_s on receipt of *preimage*$_{txid}$ does not forward it any further (Line 40).

R2RB Bootstrap protocol (Algorithm 9) describes the operations during the bootstrap phase of the R2RB protocol.

Algorithm 8: Subroutine for every node for *hold* and *pay* phase

```
def hold (hold$_x$, $Y$, $VK_{\mathrm{RN}_{(\cdot)}}$, $\nu$, $txid$, $C_{\mathrm{RN}_{(\cdot)}}$, hopMax, digest, $te_1$, $te_2$)
1   if $\nexists$ ($Y$, node$_j$, $p_j$) in routingTable with $p_j = (\mathrm{hopCount}, \mathrm{currMax}_s, \mathrm{currMax}_r, t_e)$ where
    $(\mathrm{hopCount} \leq \mathrm{hopMax}) \wedge ((\mathrm{currMax}_x \geq \nu) \vee ((\mathrm{lw}_{ij} \geq \nu) \wedge (\mathrm{currTime} \geq t_e)))$ then
2       Create tuple (holdReject$_x$, $Y$, $VK_{\mathrm{RN}_{(\cdot)}}$, $\nu$, $txid$) and send with routingTable to neighbor
        that sent hold$_x$.
3       Call multisig($Rev$,$\perp$, $i$, $j$, $\mathrm{lw}_{ij}$, $\nu$, $txid$, ts) and delete $retry.txid$, $digest.txid$,
        $segTarget.txid = \mathrm{RN}_{(\cdot)}$, $te_1.txid$, and $te_2.txid$.
4       return
5   if $\mathrm{Nonce}_i \in$ (hold$_s$, $\mathrm{Nonce}_i$, $VK_{\mathrm{RN}_{(\cdot)}}$, $\nu$, $txid$, $C_{\mathrm{RN}_{(\cdot)}}$, hopMax, digest, $te_1$, $te_2$) tuple
    belongs to node$_i$ then
6       Lookup routingTable for tuple ($\mathrm{RN}_{(\cdot)}$, node$_j$, $p_j$) with $p_j = (\mathrm{hopCount}, \mathrm{currMax}_s$,
        $\mathrm{currMax}_r, t_e)$ where $(\mathrm{hopCount} \leq \mathrm{hopMax}) \wedge ((\mathrm{currMax}_x \geq \nu) \vee ((\mathrm{lw}_{ij} \geq \nu)$
        $\wedge(\mathrm{currTime} \geq t_e)))$, update hold$_s$ tuple to (hold$_s$, $\mathrm{RN}_{(\cdot)}$, $VK_{\mathrm{RN}_{(\cdot)}}$, $\nu$, $txid$, $C_{\mathrm{RN}_{(\cdot)}}$,
        hopMax, digest, $te_1$, $te_2$) with hopMax = hopCount, and forward to node$_j$. Set
        $retry.txid = 0$, $digest.txid = digest$, $segTarget.txid = \mathrm{RN}_{(\cdot)}$,
        $te_1.txid = \mathrm{currTime} + (te_1 * \mathrm{hopCount})$, and $te_2.txid = \mathrm{currTime} + te_2$. Call
        multisig($\perp$,hold$_s$, $i$, $j$, $\mathrm{lw}_{ij}$, $\nu$,$txid$,ts).
7       Update $t_e = \mathrm{currTime} + e$ and $\mathrm{currMax}_s = \mathrm{currMax}_s - \nu$ for $p_j$ in routingTable.
        return
8   else if $node_i == \mathrm{RN}_{(\cdot)}$ then
9       Follow Alg. 6.
10  else
11      Update hold$_x$ tuple hopMax = hopMax $-$ 1 and forward tuple to node$_j$.
12      if $Y$ = $\mathrm{Nonce}_{(\cdot)}$ then
13          Set $retry.txid = 0$, $digest.txid = digest$, $segTarget.txid = \mathrm{RN}_{(\cdot)}$,
            $te_1.txid = \mathrm{currTime} + (te_1 * (\mathrm{hopCount} + \mathrm{hopMax}_{RN}))$, and
            $te_2.txid = \mathrm{currTime} + te_2$.
14      else
15          Set $retry.txid = 0$, $digest.txid = digest$, $segTarget.txid = \mathrm{RN}_{(\cdot)}$,
            $te_1.txid = \mathrm{currTime} + (te_1 * \mathrm{hopCount})$, and $te_2.txid = \mathrm{currTime} + te_2$.
16      Call multisig($\perp$,hold$_x$, $i$, $j$, $\mathrm{lw}_{ij}$, $\nu$,$txid$,ts). Update $t_e = \mathrm{currTime} + e$ and
        $\mathrm{currMax}_x = \mathrm{currMax}_x \pm \nu$ for $p_j$ in routingTable.
```

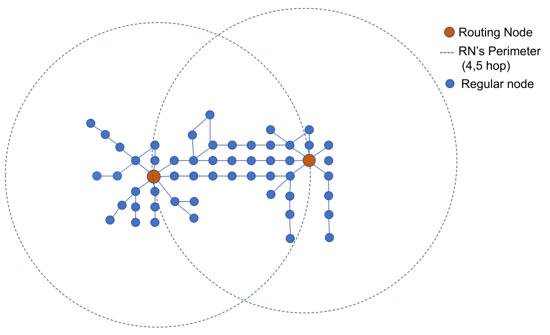


Figure 5: RNs neighborhoods in R2RB.

```
def holdReject ($holdReject_x$, $Y$, $\mathrm{VK}_{\mathrm{RN}_{(\cdot)}}$, $\nu$, $txid$)
17    Update local routingTable with new info received.
18    if ($\exists$ ($Y$, $\mathrm{node}_j$, $p_j$) with $p_j = (\mathrm{hopCount}, \mathrm{currMax}_s, \mathrm{currMax}_r, t_e)$ where $(\mathrm{hopCount} \leq \mathrm{hopMax}) \wedge (retry.txid < \mathsf{maxRetries}) \wedge ((\mathrm{currMax}_x \geq \nu) \vee ((\mathrm{lw}_{ij} \geq \nu) \wedge (\mathrm{currTime} \geq t_e)))$ then
19        Update $hold_x$ tuple $\mathrm{hopMax} = \mathrm{hopMax} - 1$ and forward tuple to $\mathrm{node}_j$. Call multisig($\perp$,$hold_x$, $i$, $j$, $\mathrm{lw}_{ij}$, $\nu$,$txid$,ts). Set $retry.txid = retry.txid + 1$.
20    else if ($\mathrm{node}_i$ == Alice $\wedge \nexists$ ($\mathrm{RN}_s$,$\mathrm{node}_j$,$p_j$)) $\vee$ ($\mathrm{node}_i$ == Bob $\wedge \nexists$ ($\mathrm{RN}_r$,$\mathrm{node}_j$,$p_j$)) where $p_j = (\mathrm{hopCount}, \mathrm{currMax}_s, \mathrm{currMax}_r, t_e)$ and $\mathrm{currMax}_x \geq \nu$ then
21        Choose new $\nu'$ and restart Algorithm 3 and 4.
22    else
23        Forward tuple ($holdReject_x$, $Y$, $VK_{\mathrm{RN}_{(\cdot)}}$, $\nu$, $txid$) along with local routingTable to neighbor that sent $hold_x$.
24        Call multisig($Rev$,$\perp$, $i$, $j$, $\mathrm{lw}_{ij}$, $\nu$,$txid$,ts) and delete $retry.txid$, $digest.txid$, $te_1.txid$, and $te_2.txid$.
def holdACK ($holdACK_x$, $t$, $\sigma_{\mathrm{RN}_{(\cdot)}}$)
25    Update local routingTable with new info received.
26    Parse $t = (txid, hold_x, \nu)$. $\mathsf{Verify}(\mathrm{vk}_{\mathrm{RN}_{(\cdot)}}, \sigma_{\mathrm{RN}_{(\cdot)}}, t) \to 1$, $\mathrm{RN}_{(\cdot)}$ == $segTarget.txid$, and delete timer $te_1$ for $txid$.
27    $\mathrm{node}_i$ then forwards the $holdACK_x$ tuple with routingTable to neighbor that sent $hold_x$.
def holdACKTimeout()
28    $\mathrm{node}_i$ calls multisig($Rev$,$\perp$, $i$, $j$, $\mathrm{lw}_{ij}$, $\nu$,$txid$,ts) to $\mathrm{node}_j$ that it had sent $hold_x$ tuple to, and retries send $hold_x$ to other neighbors for target $Y$ for $txid$. If no such neighbors exist, create $holdReject_x$ tuple, call multisig($Rev$,$\perp$, $i$, $o$, $\mathrm{lw}_{io}$, $\nu$, $txid$, ts), and send along with routingTable to $\mathrm{node}_o$ that sent $hold_x$ message. Delete $retry.txid$, $digest.txid$, $segTarget.txid = \mathrm{RN}_{(\cdot)}$, $te_1.txid$, and $te_2.txid$.
def pay($pay$, $preimage$, $\nu$, $txid$)
29    if $H(preimage) \overset{?}{\neq} digest.txid$ then
30        return $\perp$.
31    if $\mathrm{node}_i$ is an RN then
32        Follow Alg. 6.
33    if $\mathrm{node}_i$ is Alice then
34        Create $t = (pay, txid, preimage_{txid}, \nu)$, set $payACK = (payACK, t, \perp)$ and send to neighbor that sent $pay$ tuple. return
35    Forward $pay$ tuple to next neighbor $\mathrm{node}_o$ on $txid$ path along with multisig($\perp$,$pay$, $i$, $o$, $\mathrm{lw}_{io}$, $\nu$,$txid$,ts).
```

```
def payACK(payACK, ·, ·)
36   if Received $(payACK, t, \sigma_{\mathrm{RN}_{(\cdot)}})$ then
37       Parse $t = (pay, txid, \mathrm{vk}_{\mathrm{RN}_{(\cdot)}}, \nu)$, $\mathsf{Verify}(\mathrm{vk}_{\mathrm{RN}_{(\cdot)}}, \sigma_{\mathrm{RN}_{(\cdot)}}, t) \to 1$, verify $\mathrm{RN}_{(\cdot)} \in \mathbb{RN}$,
           delete $te_2.txid$.
38       $\mathrm{node}_i$ then forwards the payACK tuple to the neighbor it had received the pay tuple
           from.
39   else if Received $(payACK, t, \perp)$ then
40       Parse $t = (pay, txid, preimage_{txid}, \nu)$, verify $H(preimage_{txid} \overset{?}{=} digest.txid$, if true
           delete $te_2.txid$.
41       If $\mathrm{node}_i == \mathrm{RN}_s$, return, else forward the payACK to neighbor that sent pay tuple.
def payACKTimeout()
42   $\mathrm{node}_i$ calls multisig($Rev$,$\perp$, $i$, $j$, $\mathrm{lw}_{ij}$, $\nu$,$txid$,ts) to the neighbor $\mathrm{node}_j$ that it had originally
       sent pay tuple to and to the other neighbor that it had originally received pay tuple from.
```

We recall that R2RB differs from R2NB in the distance each RN has to broadcast the *RN-Update* message which is depicted in Figure 5. This distance is larger in R2RB because of the absence of perimeter nodes in the network. The broadcasted messages from each RN travel a certain number of hops away from the RN, allowing nodes in the given area to route transactions to the corresponding RN. Due to the larger broadcast area, neighboring RNs will receive each other's broadcast messages and be able to route transactions between them directly. The key advantage for R2RB is the elimination of perimeter nodes, with the trade-off of larger message complexity in the system due to larger broadcast distances for the *RN-Update* message.

Multisig contracts (Algorithm 10): The hold and pay phases involve neighboring nodes signing multisig contracts between them. In the hold phase, the contract stipulates that two neighboring nodes j and k agree to decrease/increase their link weights lw_{jk} and lw_{kj} respectively, by the sender's asking amount (ν) in the future when the pay tuple comes through. The multisig contract in the pay phase actually updates the link weights, and both neighboring nodes need to sign the new balances. Note that in the RN_r-Bob segment, since the payment goes in the $\mathrm{RN}_r \to$ Bob direction, the link weights are updated in the opposite direction compared to the Alice $-$ RN_s and $\mathrm{RN}_s - \mathrm{RN}_r$ segments. If a multisig contract signed in the hold or pay phases needs to be revoked, the contract and signatures on them are discarded. Algorithm 10 depicts this process in a straightforward way.

8.3 Informal Security Analysis

Bootstrapping phase: For verifying if RNs set up correct AS parameters, pp, all nodes along a path can individually check if they can produce a valid signature on a test message, else discard the pp (we have not shown this simple step for presentation clarity). If RNs do not selectively forward to certain neighbors, we do not consider it as malicious behavior. The regular nodes within a given RN's hopMax radius will receive the RN's broadcasted messages from other neighbors in the neighborhood.

The next issue is nodes underreporting or overreporting $\mathrm{currMax}_s$ and $\mathrm{currMax}_r$. We do not consider nodes underreporting $\mathrm{currMax}_s$ and $\mathrm{currMax}_r$ as malicious behavior since every node can individually decide the amount of funds to commit on its own links. If nodes overreport $\mathrm{currMax}_s$, $\mathrm{currMax}_r$ to a value greater than that of their own links, that is malicious behavior. Due to privacy concerns, nodes' link weights cannot, of course, be verified by anyone, but overreporting will eventually cause transaction failure (since there was no actual liquidity) and result in revoked hold/pay contracts with penalties for the misbehaving node. In any case, no node will lose money. The AS scheme helps verify that the $\mathrm{currMax}$ values do not increase in the series of aggregated messages to help identify malicious nodes in the network as well. A malicious node cannot increase the $\mathrm{currMax}_s$, $\mathrm{currMax}_r$ value signed

Algorithm 9: R2RB: Bootstrap broadcast from RN to RN

```
1  Each node $i$ initializes a table, *routingTable*$_i$ containing columns:
     $(\text{reachable RNs}, \text{next hop neighbor } j, \text{currMax}_s,$
     $\text{currMax}_r, \text{hopCount}, t_e)$.
2  for each RN, $k \in \mathbb{RN}$ do
3      $k$ does $\mathsf{AS.Setup}(1^\lambda) \rightarrow pp_k$ and runs $\mathsf{AS.KeyGen}(pp_k) \rightarrow (\text{sk}_k, \text{vk}_k)$.
4      Create a tuple $m_k = (\textit{RN-Update}, pp_k, \text{vk}_{RI_k}, \text{currMax}_s^k, \text{currMax}_r^k, \text{hopCount} = 0,$
         $\text{hopMax}, \mathsf{ts})$ for each neighbor $j$, $j \in [1..l]$ where $l$ is the total number of neighbors of $k$.
         Create $\sigma'_k \leftarrow \mathsf{Sign}(\text{sk}_{RI_k}, m_k)$ and set $m'_k = (m_k, \sigma'_k)$ Create signature
         $\sigma_k \leftarrow \mathsf{AS.Sign}(\text{sk}_k, \perp, \perp, \perp, m'_k)$.
5      $k$ sends $M = ((m'_k), (\text{vk}_k), \sigma_k)$ to each neighbor $j$.
6  for each node $i$ in the network on receiving an RN-Update message from neighbor $j$ do
7      On receiving $M = ((m'_k, \ldots, m_j), (\text{vk}_k, \ldots, \text{vk}_j), \sigma_j)$, $i$ parses $(m_k, \sigma'_k) \leftarrow m'_k$ and
         $(\textit{RN-Update}, pp_k, VK_k, \text{currMax}_s^k, \text{currMax}_r^k, \text{hopCount}, \text{hopMax}, \mathsf{ts}) \leftarrow m_k$.
8      if $(\mathsf{Verify}(m_k, VK_k, \sigma'_k) \rightarrow 0) \vee (\mathsf{AS.Verify}((m_k, \ldots, m_j), (\text{vk}_k, \ldots, \text{vk}_j), \sigma_j) \rightarrow 0)$ then
9          Return $\perp$.
10     $i$ checks that hopCount value in all messages $(m'_k, \ldots, m_j)$ are incremented by $1$ in each
         message. If not, return $\perp$.
11     $i$ runs $\mathsf{AS.KeyGen}(pp_k) \rightarrow (\text{sk}_i, \text{vk}_i)$.
12     $i$ updates its local *routingTable* for RN $k$ and neighbor $j$ by updating the expiry time
         $t_e = \mathsf{ts} + e$, $\text{currMax}_s^k$, and $\text{currMax}_r^k$.
13     if hopCount of received message is equal to hopMax in $m_k$ then
14         Return $\perp$.
15     else
16         for each neighbor $s$ do
17             $i$ creates $m_i$ by updating contents of $m_j$ as $\text{hopCount} = \text{hopCount} + 1$,
                 $\text{currMax}_s^k = min(\text{currMax}_s^k, \text{lw}_{i,s})$, and $\text{currMax}_r^k = min(\text{currMax}_r^k, \text{lw}_{s,i})$.
18             $i$ creates signature $\sigma_i \leftarrow \mathsf{AS.Sign}(\text{sk}_i, \sigma_j, (m'_k, \ldots, m_j), (\text{vk}_k, \ldots, \text{vk}_j), m_i)$.
19             $i$ sets $M = ((m'_k, \ldots, m_j, m_i), (\text{vk}_k, \ldots, \text{vk}_j, \text{vk}_i), \sigma_i)$ and sends it to neighbor $r$.
```

Algorithm 10: Multisig Exchange

```
Input : $o \in \{\perp, Rev\}$,$t \in \{hold_s \mid hold_r \mid pay\}$, $j$, $SK_j$, $VK_j$, $k$, $SK_k$, $VK_k$, $\mathrm{lw}_{jk}$, $\nu$,$txid$,ts
1  if $o == Rev$ then
2      $j$ and $k$ discard currently stored contracts for $txid$ and delete $fw_{jk}.txid$ and $fw_{kj}.txid$.
3      return
4  if $t == pay$ then
5      $j$ and $k$ set $\mathrm{lw}_{jk} = fw_{jk}.txid$ and $\mathrm{lw}_{kj} = fw_{kj}.txid$.
6      return
7  if $t == hold_s$ then
8      Set $fw_{jk} = \mathrm{lw}_{jk} - \nu$. Set $fw_{kj} = \mathrm{lw}_{kj} + \nu$.
9  if $t == hold_r$ then
10     Set $fw_{jk} = \mathrm{lw}_{jk} + \nu$. Set $fw_{kj} = \mathrm{lw}_{kj} - \nu$.
11 $j$ sends $\sigma_j \leftarrow \mathrm{Sign}_{SK_j}$ (contract = ($\mathrm{lw}_{jk}$,$\mathrm{lw}_{kj}$ , $fw_{jk}$, $fw_{kj}$), $txid$, digest, ts) to $k$.
12 $k$ sends $\sigma_k \leftarrow \mathrm{Sign}_{SK_k}$(contract = ($\mathrm{lw}_{jk}$,$\mathrm{lw}_{kj}$ , $fw_{jk}$, $fw_{kj}$), $txid$ , digest, ts) to $j$.
13 if $Verify_{VK_k}(\text{contract}, \sigma_k) \overset{?}{\leftarrow} 1$ then
14     $j$ stores $(\sigma_j, \sigma_k, \text{contract})$, $fw_{kj}.txid = fw_{kj}$ and $fw_{jk}.txid = fw_{jk}$.
15 if $Verify_{VK_j}(\text{contract}, \sigma_j) \overset{?}{\leftarrow} 1$ then
16     $k$ stores $(\sigma_j, \sigma_k, \text{contract})$, $fw_{kj}.txid = fw_{kj}$ and $fw_{jk}.txid = fw_{jk}$.
17 if $t == hold_s$ then
18     $j$ updates the $\mathrm{currMax}_s = min(fw_{jk}, \mathrm{currMax}_s)$ for all paths going through $k$. $k$ updates
         the $\mathrm{currMax}_r = min(fw_{kj}, \mathrm{currMax}_r)$ for all paths going through $j$.
19 if $t == hold_r$ then
20     $j$ updates the $\mathrm{currMax}_r = min(fw_{jk}, \mathrm{currMax}_r)$ for all paths going through $k$. $k$ updates
         the $\mathrm{currMax}_s = min(fw_{kj}, \mathrm{currMax}_s)$ for all paths going through $j$.
```

by the RN as part of the first aggregated message because the first message is signed by the RN using its publicly verifiable signing key.

The other potential source of malicious behavior is nodes underreporting or overreporting $\mathrm{hopCount}$ values. First note that the $\mathrm{hopCount}$ is contained in every message $((m'_k, \ldots, m_j)$, Line 7 in Algorithm 1) that is aggregated in the signature. Any honest node along a path can verify that the $\mathrm{hopCount}$ contained within every message is incremented by one, starting with $m'_k = 0$ (thus reducing $\mathrm{hopCounts}$ would be immediately detected, and the *RN-Update* message discarded). Inflating $\mathrm{hopCounts}$ would not be in the best interest of the malicious node(s) because honest nodes could have alternative shorter paths to the intended target node.

Concerning perimeter nodes, two situations could arise: Case 0: A regular node pretends to be a perimeter node by overreporting its $\mathrm{hopCount}$. In this case, that node's nonce will not figure in the set intersection of two RNs since the node was not actually a perimeter node. The node cannot do anything further. Case 1: A perimeter node underreports its hopcount or drops a message. We do not consider this malicious behavior, since it just means that the node does not wish to participate in transactions. Since the *RN-Update* messages are broadcasted, RNs will get replies from other perimeter nodes. Even if an RN does not pad its nonce list with random nonces (Algorithm 2, Line 2), it will not leak the identity of its perimeter nodes to other RNs, although it will reveal the number of perimeter nodes that RN has paths to.

Hold phase: If an honest node along a path does not receive *holdACK* or *holdReject* messages for a given transaction before the expiry of its timer te_1, the transaction will time out and will have to be retried. Malicious nodes can try to change the message type (the first field), but unknown message types will get dropped by honest nodes along a path. Malicious nodes might also try to change the "Y" parameter denoting the identity of the next RN or perimeter node to forward messages to (Algorithm 5, Case 1, 2). The message will be held at the misdirected RN/perimeter node which could also be potentially malicious. But eventually, the *hold* phase for that segment will timeout, and the *hold* contracts will be rolled back. Other parameters such as hopMax, *digest* being modified, or $C_{\mathrm{RN}(\cdot)}$ being re-encrypted (Algorithm 5, Case 1) will result in the *hold* messages being misdirected, but the *hold* phase times out, and we will not get to the *pay* phase.

A malicious RN_s cannot misroute a hold message tuple to an RN'_r instead of the sender's selected RN_r, e.g., by creating an incorrect onion. This is because Bob's *hold*$_r$ will be sent to RN_r, and since RN'_r never received it, the misrouted transaction will eventually time out, and any signed contracts will be rolled back. Similarly, no malicious node, including RNs can increase/decrease the transaction amount ν to an arbitrary value, because: 1) since the receiver knows the correct amount, the hold will eventually timeout at the last hop and fail. 2) All honest nodes along the path will have to commit to paying the amount in the hold phase. Any honest nodes which receive a *pay* message with a transaction amount different from the original *hold* message will refuse to proceed with the *pay* phase, hence timing out the transaction and causing a rollback of contracts.

The one thing that a malicious RN_s could potentially do is increase the path length to RN_r by several more RNs than is required. The transaction will eventually reach RN_r via a longer path in the $\mathrm{RN}_s - \mathrm{RN}_r$ segments. Potential solutions include the sender specifying a maximum number of layers in the onion encryption at RN_s, based on periodic network statistics released by the RNs. We leave incorporating such mechanisms into SPRITE as future work.

Pay phase: If a node intentionally misroutes the pay tuple or does not forward it, resulting in the pay tuple not reaching the target node on time, te_2 timer will expire, causing nodes to time out and rollback their pay contracts. In case of any other malicious activity, the *hold* contract signed in the previous phase can be enforced.

8.4 UC Analysis and Proof

We now prove Theorem 5.1. We give a series of games, each of which is indistinguishable from its predecessor by a PPT $\mathcal{Z}$.

Game 0: This is the same as the real-world SPRITE. $\mathcal{Z}$ interacts directly with SPRITE and $\mathcal{A}$.

Game 1: $\mathcal{S}$ internally runs $\mathcal{A}$ and simulates the secure and authenticated channels functionality $\mathcal{F}_{\mathsf{smt}}$.

Lemma 8.1. *For all PPT adversaries $\mathcal{A}$ and PPT environments $\mathcal{Z}$, there exists a simulator $\mathcal{S}$ such that*

$$\mathsf{Exec}_{\mathsf{Game0},\mathcal{Z}} \approx \mathsf{Exec}_{\mathsf{Game1},\mathcal{Z}}$$

The two games are trivially indistinguishable since $\mathcal{S}$ just executes the simulator for $\mathcal{F}_{\mathsf{smt}}$.

Game 2: $\mathcal{S}$ communicates with the honest parties and $\mathcal{A}$, and simulates the protocols of SPRITE with the help of $\mathcal{F}_{\text{sprite}}$. $\mathcal{A}$ can corrupt any user in the network, the sender, the receiver, and any or all RNs at any point in time by sending a message "corrupt" to them (although not *all* users in the network). Once an entity is corrupted, all their information is sent to $\mathcal{A}$ and all further communication to and from the corrupted party is routed through $\mathcal{A}$. We note that $suid$ depicted in Figure 8 is in lieu of sid depicted in [16]. We now state and prove the following lemma:

Lemma 8.2. *For all PPT adversaries $\mathcal{A}$ and PPT environments $\mathcal{Z}$, there exists a simulator $\mathcal{S}$ such that*

$$\mathsf{Exec}_{\mathsf{Game1},\mathcal{Z}} \approx \mathsf{Exec}_{\mathsf{Game2},\mathcal{Z}}$$

Proof. The $\mathcal{F}_{\text{setup}}$ function $\mathsf{SystemParamSetup}$ is called by $\mathcal{S}$ to setup the system parameters for SPRITE. All nodes in the real-world network establish payment channels with each other by writing the payment channel initiation transaction on the blockchain. $\mathcal{S}$ receives this information and for each honest node pair node_i and node_j, $\mathcal{S}$ creates tuples $(\mathsf{NeighborSetup}, VK_i, \mathrm{lw}_{ij}, \sigma_i)$ and $(\mathsf{NeighborSetup},\ VK_j,\ \mathrm{lw}_{ji},\ \sigma_j)$ and sends them to $\mathcal{F}_{\text{setup}}$ who adds this information to nTable. All corrupt nodes are handled by $\mathcal{A}$. All RN in $\mathbb{RN}$ register publicly as RN nodes in the real world. To simulate this, $\mathcal{S}$ sends $(\mathsf{rnRegister},\ VK_i, \sigma_i)$ to $\mathcal{F}_{\text{setup}}$ for each RN node $\mathrm{node}_i \in \mathbb{RN}$. $\mathcal{F}_{\text{setup}}$ stores all VK_i in $\mathrm{rnTable}$. RNs can be corrupted a-priori. This is handled locally by $\mathcal{A}$.

All RN nodes then create an *RN-Update* message and broadcast it to their neighbors in the real world. In the ideal world, $\mathcal{S}$ sends $\mathsf{CreateBroadcastMsg}$ to $\mathcal{F}_{\text{setup}}$ and $\mathcal{A}$ on behalf of all nodes in $\mathbb{RN}$. The resultant messages received from $\mathcal{F}_{\text{setup}}$ are forwarded to the appropriate nodes by $\mathcal{S}$ using the $\mathsf{SendMsg}$ interface of $\mathcal{F}_{\text{setup}}$. $\mathcal{S}$ receives the *RN-Update* messages of corrupt nodes from $\mathcal{A}$.

Each node in the real world verifies the broadcast on receiving it and updates its local *routingTable*. In the ideal world $\mathcal{S}$ sends $(\mathsf{VerifyBroadcast}, \mathsf{msg})$ to $\mathcal{F}_{\text{setup}}$ to verify the msg and updates the rTable accessible to the ideal functionalities. If the hopMax value has been reached in the message received, then the real-world node is finished processing the message. Let us now consider the case where an RN k is controlled by $\mathcal{A}$ and its broadcast is received by $\mathcal{S}$. $\mathcal{A}$ needs to construct its message in the right format, $((m'_k, \ldots, m_j), (\mathrm{vk}_k, \ldots, \mathrm{vk}_j, \sigma_j))$. $\mathcal{S}$ will pass on this tuple to $\mathcal{F}_{\text{setup}}$ who will verify the signature, and if VK_k has been included in rTable. $\mathcal{F}_{\text{setup}}$ also checks if the $\mathrm{hopCount}$ has been incremented by one in all consecutive messages, and will return $\perp$ if any checks fail. The nodes in $(\mathrm{vk}_k, \ldots, \mathrm{vk}_j)$ are simulated (or are clones created) by $\mathcal{A}$. This captures our real-world adversary model in that nodes can cause path inflation if they want to, but we do not check for that.

If the real-world node is within the corresponding RN's $\mathrm{hopBand}$ distance from the perimeter, i.e., if $(\mathrm{hopMax} - \mathrm{hopBand}) < \mathrm{hopCount})$, then it replies back towards the RN with a *RN-UpdateReply* message containing a Nonce before updating the received *RN-Update* message and forwarding it to the next hop in the neighborhood. To simulate this in the ideal world, $\mathcal{S}$ calls $(\mathsf{CreateNxtBroadcastMsg}, \mathsf{msg})$ where $\mathcal{F}_{\text{setup}}$ creates the updated messages to be forwarded by the node and also creates the *RN-UpdateReply*

Functionality $\mathcal{F}_{\text{setup}}$

SystemParamSetup: Upon receiving $t = (sid, \text{hopMax}, \text{pathStretch}, \text{hopBand}, \mathsf{ts}, te_1, te_2, H : \{0,1\}^* \to \{0,1\}^\lambda)$, send t to $\mathcal{S}$.

NeighborSetup: Upon receiving request $(\mathsf{NeighborSetup}, VK_i, \text{lw}_{ij}, \sigma_i, sid)$ from node_i and $(\mathsf{NeighborSetup}, VK_j, \text{lw}_{ji}, \sigma_j)$ from node_j, $\mathcal{F}_{\text{setup}}$ checks nTable for existence of tuple $(VK_i, VK_j, \cdot, \cdot, \cdot, \cdot)$, if so, return $\perp$, or continue. $\mathcal{F}_{\text{setup}}$ internally calls $\mathcal{F}_{\text{sig}}$ to verify σ_i and σ_j. If $\mathcal{F}_{\text{sig}}$ returns 1 for both checks then $\mathcal{F}_{\text{setup}}$ stores $(VK_i, VK_j, \text{lw}_{ij}, \text{lw}_{ji}, \sigma_i, \sigma_j)$ in nTable and returns $(\mathsf{success}, VK_i, VK_j)$ to node_i and node_j.

rhRegister: Upon receiving request $(\mathsf{rnRegister}, VK_i, \sigma_i, sid)$, $\mathcal{F}_{\text{setup}}$ internally calls $\mathcal{F}_{\text{sig}}$ to verify σ_i. If $\mathcal{F}_{\text{sig}}$ returns 1 then $\mathcal{F}_{\text{setup}}$ stores VK_i in rnTable and returns "$\mathsf{success}$" to node_i.

SendMessage: Upon receiving request $(\mathsf{SendMsg}, \mathsf{msg}, [\text{node}_k, \ldots, \text{node}_l], sid)$ from node_i, $\mathcal{F}_{\text{setup}}$ checks that each recipient $[node_k, \ldots, \text{node}_l]$ is a neighbor of node_i according to nTable, if verification fails then return $\perp$ to node_i. Forward msg to each neighbor in list. **createBroadcast**: Upon receiving request $(\mathsf{CreateBroadcastMsg}, sid)$ from node_i, $\mathcal{F}_{\text{setup}}$ checks if tuple VK_i in rnTable. If not, return $\perp$, else continue. For each neighbor of node_i according to nTable create m_i = (*RN-Update*, pp_i, VK_i, currMax_s^i, currMax_r^i, $\text{hopCount} = 0$, hopBand, hopMax, $\mathsf{ts} \in \mathbb{R}^+$) where currMax_s^i and currMax_r^i are the link weights between node_i and the respective neighbor. It then calls $\mathcal{F}_{\text{sig}}$ to generate signature σ_i' on m_i and sets $m_i' = (m_i, \sigma_i')$. It generates string $\sigma_{\text{agg}}, \text{vk}_i \leftarrow\!\!\$ \{0,1\}^k$, $(m_i', VK_i, \sigma_{\text{agg}})$ to sTable, and returns (*RN-Update*, m_i', vk_i, σ_{agg}) to $\mathcal{S}$.

verifyBroadcast: Upon receiving request $(\mathsf{VerifyBroadcast}, \mathsf{msg}, sid)$ from node_i or $\mathcal{S}$, $\mathcal{F}_{\text{setup}}$ parses $((m_k', \ldots, m_j), (\text{vk}_k, \ldots, \text{vk}_j), \sigma_j) \leftarrow \mathsf{msg}$, $(m_k, \sigma_k') \leftarrow m_k'$ and (*RN-Update*, pp_k, VK_k, currMax_s^k, currMax_r^k, hopCount, hopBand, hopMax, ts) $\leftarrow m_k$. $\mathcal{F}_{\text{setup}}$ sends $(\mathsf{Verify}, uid, m, \sigma, pk')$ to $\mathcal{F}_{\text{sig}}$, who returns $(\mathsf{Verified}, uid, m, f)$, $f \in \{0, 1, \phi\}$. If verification fails or tuple $\text{VK}_k \nexists$ rnTable, then return $\perp$ to node_i or $\mathcal{S}$, else continue. $\mathcal{F}_{\text{setup}}$ checks if tuple $(m_k', \ldots, m_j, VK_k, \ldots, \text{vk}_j, \sigma_j)$ exists in sTable and check if the hopCount in each message in $m_k', \ldots, m_j$ is incremented by 1. If not found return $\perp$ to the calling entity and $\mathcal{S}$, else $\mathcal{F}_{\text{setup}}$ adds tuple $(i, VK_k, \text{node}_j, \text{currMax}_s^k, \text{currMax}_r^k, \text{hopCount}, \mathsf{ts})$ in table rTable. and return "accept" to node_i.

decrypt: Upon receiving request $(\mathsf{decrypt}, \text{cTxt}, sid)$ from node_i, $\mathcal{F}_{\text{setup}}$ checks if a tuple $(\text{cTxt}, \text{pTxt}, i)$ exists in cTable. If so, $\mathcal{F}_{\text{setup}}$ returns pTxt to node_i, else return $\perp$.

createNextBcastMsg: Upon receiving request $(\mathsf{CreateNxtBroadcastMsg}, \mathsf{msg}, sid)$ from node_i, $\mathcal{F}_{\text{setup}}$ parses $((m_k', \ldots, m_j), (\text{vk}_k, \ldots, \text{vk}_j), \sigma_j) \leftarrow \mathsf{msg}$. $(m_k, \sigma_k') \leftarrow m_k'$ and (*RN-Update*, pp_k, vk_{RI_k}, currMax_s^k, currMax_r^k, hopCount, hopBand, hopMax, ts) $\leftarrow m_j$. $\mathcal{F}_{\text{setup}}$ calls $\mathcal{F}_{\text{sig}}$ with (KeyGen, uid). When $\mathcal{F}_{\text{sig}}$ returns $(\mathsf{VerificationKey}, uid, pk_i)$, $\mathcal{F}_{\text{setup}}$ records the pair (i, pk_i) in kTable. If (hopCount $>$ hopMax), return $\perp$ to $\mathcal{S}$ (to $\mathcal{S}$ or node_i). $\mathcal{F}_{\text{setup}}$ initializes list of messages $M_i = []$. For each neighbor tuple $(VK_{\text{node}_i}, VK_{\text{node}_l}, \text{lw}_{lk}, \text{lw}_{kl}, \sigma_i, \sigma_l)$ in nTable, $\mathcal{F}_{\text{setup}}$ creates a message m_{il} = (*RN-Update*, pp_k, vk_{RI_k}, currMax_s^k, currMax_r^k, hopCount $+ 1$, hopBand, hopMax, ts), where $\text{currMax}_s^k = min(\text{currMax}_s^k, \text{lw}_{li})$ and $\text{currMax}_r^k = min(\text{currMax}_r^k, \text{lw}_{il})$. $\mathcal{F}_{\text{setup}}$ generates a string $\sigma_{\text{agg}} \leftarrow\!\!\$ \{0,1\}^k$, adds tuple $(m_k', \ldots, m_j, m_{il}, \text{vk}_k, \ldots, \text{vk}_j, \text{vk}_i, \sigma_{\text{agg}})$ to sTable and adds (*RN-Update*, $m_k', \ldots, m_j, m_{il}, \text{vk}_k, \ldots, \text{vk}_j, \text{vk}_i, \sigma_{\text{agg}}$) to M_i. If ((hopMax $-$hopBand) $<$ hopCount) $\wedge$ (hopCount $\leq$ hopMax), $\mathcal{F}_{\text{setup}}$ retrieves tuple $(VK_i, VK_j, \text{lw}_{ij}, \text{lw}_{ji}, \sigma_i, \sigma_j)$ from nTable, creates nonce, $\text{Nonce}_i \leftarrow\!\!\$ \{0,1\}^\lambda$, creates message mr_i' = (*RN-UpdateReply*, pp_k, vk_{RI_k}, currMax_s^k, currMax_r^k, hopCount $+ 1$, hopBand, hopMax, ts,Nonce_i) where $\text{currMax}_s^k = min(\text{currMax}_s^k, \text{lw}_{j,i})$ and $\text{currMax}_r^k = min(\text{currMax}_r^k, \text{lw}_{i,j})$. $\mathcal{F}_{\text{setup}}$ generates a string $\sigma_{\text{agg}} \leftarrow\!\!\$ \{0,1\}^k$, adds tuple $(m_k', \ldots, m_j, mr_i', \text{vk}_k, \ldots, \text{vk}_j, \text{vk}_i, \sigma_{\text{agg}})$ to sTable and adds (*RN-Update*, $m_k', \ldots, m_j, mr_i', \text{vk}_k, \ldots, \text{vk}_j, \text{vk}_i, \sigma_{\text{agg}}$) to M_i. $\mathcal{F}_{\text{setup}}$ sends M_i to $\mathcal{S}$.

Figure 6: Ideal functionality for system setup and network bootstrap (continued)

VerCreateNextRNUpRep: Upon receiving request $(\mathsf{VerCreateNextRNUpRep}, \mathsf{msg}, sid)$ from node_i, $\mathcal{F}_{\text{setup}}$ parses $(\textit{RN-UpdateReply}, m'_k, \ldots, mr'_o, \text{vk}_k, \ldots, \text{vk}_o, \sigma_o) \leftarrow \mathsf{msg}$, checks if a tuple $(m'_k, \ldots, mr'_o, \text{vk}_k, \ldots, \text{vk}_o, \sigma_o)$ exists in sTable, if so continue, else return $\perp$ to $\mathcal{S}$. If tuple $VK_i \in$ rnTable, $\mathcal{F}_{\text{setup}}$ checks if $(\text{hopMax} - \text{hopBand}) \overset{?}{\leq} |m'_i, \ldots, mr'_o|/2 \overset{?}{\leq} \text{hopMax})$, if so continue, else return $\perp$ to $\mathcal{S}$. $\mathcal{F}_{\text{setup}}$ parse $(\textit{RN-UpdateReply}, pp_i, \text{vk}_{RI_i}, \text{currMax}^i_s, \text{currMax}^i_r, \text{hopCount}, \text{hopBand}, \text{hopMax}, \mathsf{ts}, \text{Nonce}_l) \leftarrow mr'_o$, add tuple $(i, \text{Nonce}_l, \text{node}_o, \text{currMax}^i_s, \text{currMax}^i_r, \text{hopCount}, \mathsf{ts})$ to rTable and return "accept" to node_i. If tuple $VK_i \notin$ rnTable, then update mr'_o as $mr'_i = (\textit{RN-UpdateReply}, pp_k, VK_k, \text{currMax}^k_s, \text{currMax}^k_r, \text{hopCount} - 1, \text{hopBand}, \text{hopMax}, \mathsf{ts}, \text{Nonce}_l)$, $\mathcal{F}_{\text{setup}}$ generates a string $\sigma_{\text{agg}} \leftarrow\$ \{0,1\}^k$, adds tuple $(m'_k, \ldots, mr'_o, mr'_i, \text{vk}_k, \ldots, \text{vk}_o, \text{vk}_i, \sigma_{\text{agg}})$ to sTable, add $(\text{node}_i, \text{Nonce}_l, \text{node}_j, \text{currMax}^k_s, \text{currMax}^k_r, \text{hopCount}, \mathsf{ts})$ in table rTable, and return $(\textit{RN-UpdateReply}, m'_i, \ldots, mr'_o, mr'_i, \text{vk}_k, \ldots, \text{vk}_o, \text{vk}_i, \sigma_{\text{agg}})$ to $\mathcal{S}$.
populateRHRTable: Upon receiving request $(\mathsf{populateRHRTable}, sid)$ from node_i, $\mathcal{F}_{\text{setup}}$ checks if tuple $VK_i \in$ rnTable, if so, continue, else return $\perp$. It then retrieves each tuple $(j, \text{Nonce}_l, \cdot, \cdot, \cdot, \cdot, \cdot)$ in rTable for each node_j with $VK_j \in$ rnTable, and adds Nonce_l to tuple $(j, \cdot, [\cdot, \cdot, \text{Nonce}_l])$ in rnrTable. $\mathcal{F}_{\text{setup}}$ then computes the second column of rnrTable by checking common Nonces of nodes in rnrTable, e.g., if j in rnrTable has common nonces with k, l, m in rnrTable, $\mathcal{F}_{\text{setup}}$ updates j's tuple as $(j, [k, l, m], [[\text{Nonce}_{(\cdot)}]_k, [\text{Nonce}_{(\cdot)}]_l, [\text{Nonce}_{(\cdot)}]_m])$. $\mathcal{F}_{\text{setup}}$ then sends all tuples of rnrTable in the form of $(\cdot, \cdot, \perp)$ to all nodes o where $\text{VK}_o \in$ rnTable.

Figure 6: Ideal functionality for system setup and network bootstrap

if the current node is within the RN's hopBand distance from the perimeter. Now corrupt nodes simulated by $\mathcal{A}$ can choose not to reply.

The creation of aggregate signatures populates tuples in sTable which the functionalities can check for verification of the aggregate signatures. In the real world, on reception of *RN-UpdateReply* message, nodes update their *routingTable* and forward the message back towards the RN. In the ideal world, $\mathcal{S}$ sends $(\mathsf{VerCreateNextRNUpRep}, \mathsf{msg})$ to $\mathcal{F}_{\text{setup}}$ on receiving the *RN-UpdateReply* message msg from a node. $\mathcal{F}_{\text{setup}}$ verifies the contents of the message, updates rTable for the node and replies back with the updated message to be forwarded by the said node to its neighbors towards RN. Now the next node could be simulated by $\mathcal{A}$. First off, if the message tuple $(m'_k, \ldots, m'_{r_o}, \cdot, \cdot)$ does not exist in sTable, that means $\mathcal{A}$ has responded to an *RN-Update* request that was never sent out, and the simulation aborts. If the hop conditions are not satisfied, $\mathcal{A}$ is returned $\perp$. Else $\mathcal{A}$'s reply is accepted and the rTable is correspondingly updated by $\mathcal{F}_{\text{setup}}$. If $\mathcal{A}$ does not respond the simulation times out and aborts.

In the real world, RNs on receiving replies for all their broadcast messages share the Nonces received and create *RNroutingTable* to be able to route transactions through other RNs. In the ideal world, $\mathcal{S}$ sends $\mathsf{populateRNRTable}$ to $\mathcal{F}_{\text{setup}}$ which evaluates rTable entries for all RNs and populates rnrTable with routing information to help RNs route messages between them. If any corrupt RN is simulated by $\mathcal{A}$, it get its information by querying $\mathcal{F}_{\text{setup}}$ for the appropriate entries in the rTable, which will be given correctly to $\mathcal{A}$. For a subset of RNs that are simulated by $\mathcal{A}$, the set intersection to find perimeter nodes is done locally. For honest perimeter nodes between RNs, $\mathcal{S}$ would have passed on the information to $\mathcal{F}_{\text{setup}}$, who would have computed and updated the rTable correctly. If all RNs are controlled by $\mathcal{A}$, the simulation is handled locally by $\mathcal{A}$.

Functionality $\mathcal{F}_{\text{hold}}$

createHoldSend: $\mathcal{F}_{\text{hold}}$ on receiving $(\text{create}\textit{hold}_s, \nu, \text{RN}_s, \text{RN}_r, \textit{sid})$ from node_i checks if tuple $(i, \text{RN}_s, j, \text{currMax}_s^k, \text{currMax}_r^k, \text{hopCount}, \text{ts})$ exists in rTable with $\text{currMax}_s^k \geq \nu$, if so continue, else return $\perp$. $\mathcal{F}_{\text{hold}}$ does $\textit{preimage}, \textit{preimage}_{\textit{txid}}, \textit{token} \leftarrow\$ \{0,1\}^\lambda$, $\textit{digest} = H(\textit{preimage})$, $\textit{txid} = H(\textit{preimage}_{\textit{txid}})$, $\text{cTxt}_{\text{RN}_s}, \text{cTxt}_{\text{RN}_r} \leftarrow\$ \{0,1\}^\lambda$, adds tuples $(\text{cTxt}_{\text{RN}_r}, (\textit{token}, \nu, \textit{txid}), \text{RN}_r)$ and $(\text{cTxt}_{\text{RN}_s}, (VK_{\text{RN}_r}, \nu, \textit{txid}, \text{cTxt}_{\text{RN}_r}), \text{RN}_s)$ to cTable. $\mathcal{F}_{\text{hold}}$ adds $(\textit{txid}, \textit{preimage}_{\textit{txid}}, \textit{digest}, \textit{preimage}, \textit{token})$ to txidTable. $\mathcal{F}_{\text{hold}}$ then returns $(\textit{hold}_s, \text{RN}_s, VK_{\text{RN}_s}, \nu, \textit{txid}, \text{cTxt}_{\text{RN}_s}, \text{hopMax}, \textit{digest}, te_1, te_2)$ and node_j to node_i.

createHoldRecv: $\mathcal{F}_{\text{hold}}$ on receiving $(\text{create}\textit{hold}_r, \textit{token}, \nu, \textit{txid}, \textit{digest}, \text{RN}_r, \textit{sid})$ from node_i checks if tuple $(i, \text{RN}_r, j, \text{currMax}_s^k, \text{currMax}_r^k, \text{hopCount}, \text{ts})$ exists in rTable with $\text{currMax}_r^k \geq \nu$, if so continue, else return $\perp$. $\mathcal{F}_{\text{hold}}$ creates $\text{cTxt}'_{\text{RN}_r} \leftarrow\$ \{0,1\}^\lambda$, adds tuple $(\text{cTxt}'_{\text{RN}_r}, (\textit{token}, \nu, \textit{txid}), \text{RN}_r)$ to cTable, and returns tuple $(\textit{hold}_r, \text{RN}_r, VK_{\text{RN}_r}, \nu, \textit{txid}, \text{cTxt}'_{\text{RN}_r}, \text{hopMax}, \textit{digest}, te_1, te_2)$ and node_j to node_i.

verifyandCreateNextHold: $\mathcal{F}_{\text{hold}}$ on receiving $(\text{verifyandcreatenext}\textit{hold}_{(\cdot)}, \textit{msg}, \textit{sid})$ from node_i, parses $\textit{msg} = (\textit{hold}_{(\cdot)}, Y, VK_{\text{RN}_{(\cdot)}}, \nu, \textit{txid}, \text{cTxt}_{(\cdot)}, \text{hopMax}, \textit{digest}, te_1, te_2)$. If Nonce belongs to node_i, $\mathcal{F}_{\text{hold}}$ updates $\mathsf{msg} = (\textit{hold}_{(\cdot)}, \text{RN}_{(\cdot)}, VK_{\text{RN}_{(\cdot)}}, \nu, \textit{txid}, \text{cTxt}_{(\cdot)}, \text{hopMax}, \textit{digest}, te_1, te_2)$. $\mathcal{F}_{\text{hold}}$ checks if tuple $(i, \text{RN}_{(\cdot)}, j, \text{currMax}_s^k, \text{currMax}_r^k, \text{hopCount}, \text{ts})$ exists in rTable and satisfies the requirements of $\textit{msg}$ (liquidities are satisfied, etc.) and if so updates $\text{hopMax} = \text{hopMax} - 1$ in msg and returns $(\text{node}_j, \textit{msg})$ to node_i. If the information in rTable is expired $(\text{currTime} \geq t_e)$, check $(\text{node}_i, \text{RN}_{(\cdot)}, \text{node}_j, \cdot, \cdot, \text{hopCount}, \text{ts})$ exists, if so updates $\text{hopMax} = \text{hopMax} - 1$ in msg and return $(\text{node}_j, \textit{msg})$ to node_i. $\mathcal{F}_{\text{hold}}$ adds tuple $(i, \textit{txid}, 0, te_1.\textit{txid} = \text{currTime} + (te_1 * \text{hopCount}), te_2.\textit{txid} = \text{currTime} + (te_2 * \text{hopCount}))$ to txTeTable. If $(\textit{hold}_{(\cdot)}, Y, \text{RN}_{(\cdot)}, \text{VK}_{\text{RN}_{(\cdot)}}, \cdot, \textit{txid}, \cdot, \cdot, \cdot, \cdot, \cdot)$ then if any of the checks fail, return $\perp$ to node_i.

verifyandCreateNextHoldRH: $\mathcal{F}_{\text{hold}}$ on receiving $(\text{verifyandcreatenextRN}\textit{hold}_{(\cdot)}, \textit{msg}, \textit{sid})$ from node_i, parses $\textit{msg} = (\textit{hold}_{(\cdot)}, \text{RN}_{(\cdot)}, VK_{\text{RN}_{(\cdot)}}, \nu, \textit{txid}, \text{cTxt}_{(\cdot)}, \text{hopMax}, \textit{digest}, te_1, te_2)$, and checks if tuple $VK_i \in$ rnTable. If not then return $\perp$, else continue. $\mathcal{F}_{\text{hold}}$ calls $\mathcal{F}_{\text{sig}}$ function $(\text{Sign}, \text{RN}(\cdot), (\textit{txid}, \textit{hold}_{(\cdot)}, \nu))$ and adds the return σ to a message $(\textit{holdACKs}, (\textit{txid}, \textit{hold}_{(\cdot)}, \nu), \sigma)$ and sends it to node_i.

- If $\text{RN}_{(\cdot)} == \text{RN}_s$, $\mathcal{F}_{\text{hold}}$ retrieves $(\text{cTxt}_{\text{RN}_s}, \text{pTxt}, \text{RN}_s)$ from cTable, parses $\text{pTxt} = (\text{vk}_{\text{RN}_r}, \nu, \textit{txid}, \text{cTxt}_{\text{RN}_r})$ and generates ciphertexts for onion type encryption for each successive RN till RN_r according to rnrTable by finding RNs with common Nonces in their tuples. With $\text{cTxt}_{\text{RN}_o}$ being the ciphertext for the neighboring RN of RN_s and outermost layer of the onion encryption, $\mathcal{F}_{\text{hold}}$ creates message $(\textit{hold}_{(\cdot)}, \text{Nonce}, VK_{\text{RN}_o}, \nu, \textit{txid}, \text{cTxt}_{\text{RN}_o}, \text{hopMax}, \textit{digest}, te_1, te_2)$ where Nonce is common in tuples for node_i and RN_o according to rTable, and sends to node_i or $\mathcal{S}$. If no such tuples exist, return $\perp$.

- Else if $\text{RN}_{(\cdot)} == \text{RN}_r$ and message type is $\text{verifyandcreatenextRN}\textit{hold}_s$ then $\mathcal{F}_{\text{hold}}$ retrieves $(\text{cTxt}_{\text{RN}_r}, \text{pTxt}, \text{RN}_r)$ from cTable, parses $\text{pTxt} = (\textit{token}, \nu, \textit{txid})$ and stores $\textit{token}, \textit{txid}, \textit{send}$ internally. If $\text{RN}_{(\cdot)} == \text{RN}_r$ and message type is $\text{verifyandcreatenextRN}\textit{hold}_r$, then $\mathcal{F}_{\text{hold}}$ retrieves $(\text{cTxt}'_{\text{RN}_r}, \text{pTxt}, \text{RN}_r)$, parses $\text{pTxt} = (\textit{token}, \nu, \textit{txid})$ and stores $\textit{token}, \textit{txid}, \textit{recv}$ internally. If $\mathcal{F}_{\text{hold}}$ has $(\textit{token}, \textit{txid}, \textit{send})$ and $(\textit{token}, \textit{txid}, \textit{recv})$, then it sends $(\textit{proceedPay}, \textit{txid}, \nu, \sigma_{\textit{proceedPay}})$, where $\sigma_{\textit{proceedPay}}$ is the signature on tuple $(\textit{proceedPay}, \textit{txid}, \nu)$, to $\mathcal{A}$ or $\mathcal{S}$.

- Else $\mathcal{F}_{\text{hold}}$ retrieves $(\text{cTxt}_{\text{node}_i}, \text{pTxt}, \text{node}_i)$ from cTable, parses $\text{pTxt} = (\text{vk}_{\text{RN}_o}, \nu, \textit{txid}, \text{cTxt}_{\text{RN}_o})$, creates next message $(\textit{hold}_{(\cdot)}, \text{Nonce}, VK_{\text{RN}_o}, \nu, \textit{txid}, \text{cTxt}_{\text{RN}_o}, \text{hopMax}, \textit{digest}, te_1, te_2)$ according to the rTable by finding a common Nonce in tuples for node_i and RN_o using rnrTable. If no such tuples exist, return $\perp$.

Figure 7: Ideal functionality for hold phase (continued)

multisig: $\mathcal{F}_{\text{hold}}$ on receiving $(\text{multisig}, \sigma_j, \text{node}_j, \text{lw}_{jk}, \text{lw}_{kj}, fw_{jk}, fw_{kj}, txid, \textit{digest}, \textsf{ts}, sid)$ from node_j and $(\text{multisig}, \sigma_k, \text{node}_k, \text{lw}_{jk}, \text{lw}_{kj}, fw_{jk}, fw_{kj}, txid, \textit{digest}, \textsf{ts}, sid)$ from node_k, calls $\mathcal{F}_{\text{sig}}$ to verify the σ_k and σ_j on ($\textsf{contract} = \text{lw}_{jk}, \text{lw}_{kj}, fw_{jk}, fw_{kj}, txid, \textit{digest}, \textsf{ts}$), and check if tuple $(VK_j, VK_k, \cdot, \cdot, \cdot,\cdot)$ exists in nTable. If all verifications pass, then $\mathcal{F}_{\text{hold}}$ stores $\textsf{contract}_{txid}$ and updates the appropriate entries for node_j and node_k in rTable, $\text{currMax}_x^{(\cdot)}$ to $\text{currMax}_x^{(\cdot)} \pm \nu$ and returns "success" to node_j and node_k. Else return $\perp$ to node_j and node_k.
multisigRev: $\mathcal{F}_{\text{hold}}$ on receiving $(\text{multisigRev}, \sigma_j, \text{node}_j, \text{lw}_{jk}, \text{lw}_{kj}, fw_{jk}, fw_{kj}, txid, \textit{digest}, \textsf{ts}, sid)$ from node_j and $(\text{multisigRev}, \sigma_k, \text{node}_k, \text{lw}_{jk}, \text{lw}_{kj}, fw_{jk}, fw_{kj}, txid, \textit{digest}, \textsf{ts}, sid)$ from node_k, $\mathcal{F}_{\text{hold}}$ checks if tuple $(VK_j, VK_k, \cdot, \cdot, \cdot,\cdot)$ exists in nTable. If so, it deletes stored $\textsf{contract}_{txid}$ and updates the appropriate entries for node_j and node_k in rTable, $\text{currMax}_x^{(\cdot)}$ to $\text{currMax}_x^{(\cdot)} \pm \nu$ and returns "success" to node_j and node_k. Else return $\perp$.
createHoldReject: $\mathcal{F}_{\text{hold}}$ on receiving $(\text{createHoldRej}, msg, sid)$ from node_i, parses $msg = (\textit{hold}_{(\cdot)}, \cdot, VK_{\text{RN}_{(\cdot)}}, \nu, txid, \text{cTxt}_{(\cdot)}, \text{hopMax}, \textit{digest}, te_1, te_2)$, creates tuple $(\textit{holdReject}_{(\cdot)}, \cdot, VK_{\text{RN}_{(\cdot)}}, \nu, txid)$, retrieves all tuples $(i, VK_{\text{RN}_{(\cdot)}}, \cdot, \cdot, \cdot, \cdot, \cdot)$ from rTable, and sends the $\textit{holdReject}_{(\cdot)}$ tuple along with rnTable tuples to node_i.
verifyHoldReject: $\mathcal{F}_{\text{hold}}$ on receiving $(\text{verifyHoldReject}, msg, sid)$ from node_i, update the currMax_s and currMax_r values from rTable tuples received in msg as appropriate.
verifyHoldAck: $\mathcal{F}_{\text{hold}}$ on receiving $(\text{verifyHoldAck}, msg, sid)$ from node_i, update the currMax_s and currMax_r values from rTable tuples received in msg as appropriate.

Figure 7: Ideal functionality for hold phase

Functionality $\mathcal{F}_{\text{sig}}$

Key Generation: Upon receiving a value $(\textsf{KeyGen}, \text{suid})$ from some party S, verify that $\text{suid} = (S, \text{suid}')$ for some suid'. If not, then ignore the request. Else, hand $(\textsf{KeyGen}, \text{suid})$ to the adversary. Upon receiving $(\textsf{VerificationKey}, \text{suid}, v)$ from the adversary, output $(\textsf{VerificationKey}, \text{suid}, v)$ to S, and record the pair (S, v).
Signature Generation: Upon receiving a value $(\text{Sign}, \text{suid}, m)$ from S, verify that $\text{suid} = (S, \text{suid}')$ for some suid'. If not, then ignore the request. Else, send $(\text{Sign}, \text{suid}, m)$ to the adversary. Upon receiving $(\text{Signature}, \text{suid}, m, \sigma)$ from the adversary, verify that no entry $(m, \sigma, v, 0)$ is recorded. If it is, then output an error message to S and halt. Else, output $(\text{Signature}, \text{suid}, m, \sigma)$ to S, and record the entry $(m, \sigma, v, 1)$.
Signature Verification: Upon receiving a value $(\text{Verify}, \text{suid}, m, \sigma, v_0)$ from some party P, hand $(\text{Verify}, \text{suid}, m, \sigma, v_0)$ to the adversary. Upon receiving $(\text{Verified}, \text{suid}, m, \varphi)$ from the adversary do:
1) If $v_0 = v$ and the entry $(m, \sigma, v, 1)$ is recorded, then set $f = 1$. (This condition guarantees completeness: If the verification key v_0 is the registered one and σ is a legitimately generated signature for m, then the verification succeeds.)
2) Else, if $v_0 = v$, the signer is not corrupted, and no entry $(m, \sigma_0, v, 1)$ for any σ_0 is recorded, then set $f = 0$ and record the entry $(m, \sigma, v, 0)$. (This condition guarantees unforgeability: If v_0 is the registered one, the signer is not corrupted, and never signed m, then the verification fails.)
3) Else, if there is an entry (m, σ, v_0, f_0) recorded, then let $f = f_0$. (This condition guarantees consistency: All verification requests with identical parameters will result in the same answer.)
4) Else, let $f = \varphi$ and record the entry $(m, \sigma, v_0, \varphi)$.
Output $(\text{Verified}, \text{id}, m, f)$ to P.

Figure 8: Ideal functionality for signature generation and verification [16]

$\mathcal{S}$ sends $(\text{create}\textit{hold}_s, \nu, \text{RN}_s, \text{RN}_r)$ to $\mathcal{F}_{\text{hold}}$ on behalf of a sender (node_i) to $\mathcal{F}_{\text{hold}}$ to send ν credits through RN_s to RN_r. $\mathcal{F}_{\text{hold}}$ returns a message $(\textit{hold}_s, \text{RN}_s, VK_{\text{RN}_s}, \nu,$ $\textit{txid}$, $\text{cTxt}_{\text{RN}_s}$, hopMax, *digest*, te_1, $te_2)$ to $\mathcal{S}$ which is sent to the node according to rTable tuples for the node_i. $\mathcal{S}$ sends $(\text{create}\textit{hold}_r, \textit{token}, \nu, \textit{txid}, \textit{digest}, \text{RN}_r)$ on behalf of a receiver (node_i) to $\mathcal{F}_{\text{hold}}$ to receive ν credits from RN_r. $\mathcal{F}_{\text{hold}}$ returns a message $(\textit{hold}_r, \text{RN}_r, VK_{\text{RN}_r}, \nu,$ $\textit{txid}, \text{cTxt}'_{\text{RN}_r}$, hopMax, *digest*, te_1, $te_2)$ to $\mathcal{S}$ which is sent to the node according to rTable tuples for the node_i. If sender is malicious, $\mathcal{A}$ will handle the simulation of the sender, but since per our threat model, $\mathcal{A}$ cannot corrupt *all* nodes in the PCN, there will be a point where a node controlled by $\mathcal{A}$ has to pass on a $(\textit{hold}_s, \text{RN}_s, \text{VK}_{\text{RN}_s}, \nu,$ $\textit{txid}$, $\text{cTxt}_{\text{RN}_s}$, hopMax, *digest*, te_1, $te_2)$. If any part of this tuple is malformed, it will caught by $\mathcal{F}_{\text{setup}}$ during the simulation of any (even a single) honest intermediary along the path as we describe below. The case where the receiver is controlled by $\mathcal{A}$ is handled in a similar way. Note that $\mathcal{A}$ may choose to ignore its own rTable or the information returned by $\mathcal{F}_{\text{setup}}$ and deliberately choose a longer path. This either means that $\mathcal{A}$ is increasing its own latency if it is the sender/receiver, or it is inflating the path (by a small margin). In our threat model we do not consider finding longer paths necessitated due to node disconnections/failure as malicious behavior. This is reflected in the ideal world too.

Functionality $\mathcal{F}_{\text{pay}}$

createpay: $\mathcal{F}_{\text{pay}}$ on receiving (createPay, (*proceedPay*, $\textit{txid}$, ν, $\sigma_{\textit{proceedPay}}$, $\textit{sid}$)) from node_i, calls $\mathcal{F}_{\text{sig}}$ to verify $\sigma_{\textit{proceedPay}}$, if check passes continue, else return $\perp$. $\mathcal{F}_{\text{pay}}$ retrieves tuple $(\textit{txid}, \cdot, \cdot, \textit{preimage}, \cdot)$ from txidTable and s sends $(\textit{pay}, \textit{preimage}, \nu, \textit{txid})$ to node_i.

verifyPay: $\mathcal{F}_{\text{pay}}$ on receiving (verifyPay, *preimage*, ν, $\textit{txid}$, $\textit{sid}$) from node_i, checks if $\textit{digest} \overset{?}{=} H(\textit{preimage})$, if check passes, return "success" to node_i, else return $\perp$.

verifyPayAck: $\mathcal{F}_{\text{pay}}$ on receiving (verifyPayAck, $((\textit{pay}, \textit{txid}, \text{vk}_{\text{node}_i}, \nu), \sigma_{\text{RN}_{(\cdot)}})$, $\textit{sid}$), from node_i calls $\mathcal{F}_{\text{sig}}$ to verify, $\sigma_{\text{RN}_{(\cdot)}}$ if check passes then return "success" to node_i, else return $\perp$. $\mathcal{F}_{\text{pay}}$ on receiving (verifyPayAck, $((\textit{pay}, \textit{txid}, \textit{preimage}_{\textit{txid}}, \nu), \perp)$), checks if $\textit{txid} \overset{?}{=} H(\textit{preimage}_{\textit{txid}})$, if check passes then return "success" to node_i, else return $\perp$.

verifyPayRN: $\mathcal{F}_{\text{pay}}$ on receiving (verifyPayRN, *preimage*, ν, $\textit{txid}$, $\textit{sid}$) from node_i, checks if VK_{node_i} in rnTable and if $\textit{digest} \overset{?}{=} H(\textit{preimage})$. If the checks pass, then call $\mathcal{F}_{\text{sig}}$ to create signature σ_i on $(\textit{pay}, \textit{txid}, \text{vk}_i, \nu)$ and send $(\textit{payACK}, (\textit{pay}, \textit{txid}, \text{vk}_i, \nu), \sigma_i)$ to node_i.

verifypaySender: $\mathcal{F}_{\text{pay}}$ on receiving (verifyPaySender ,*preimage*, ν, $\textit{txid}$, $\textit{sid}$), checks if $\textit{digest} \overset{?}{=} H(\textit{preimage})$. If the check passes, then $\mathcal{F}_{\text{pay}}$ creates tuple, $(\textit{payACK}, (\textit{pay}, \textit{txid}, \textit{preimage}_{\textit{txid}}, \nu), \perp)$ and sends it to node_i, else return $\perp$.

multisigPay: $\mathcal{F}_{\text{pay}}$ on receiving (multisigPay,σ_j,node_j,lw_{jk}, lw_{kj}, fw_{jk}, fw_{kj}, $\textit{txid}$, *digest*, ts, $\textit{sid}$) from node_j and (multisigPay, σ_k, node_k,lw_{jk},lw_{kj}, fw_{jk},fw_{kj}, $\textit{txid}$, *digest*, ts, $\textit{sid}$) from node_k, $\mathcal{F}_{\text{pay}}$ verifies σ_j and σ_k by calling $\mathcal{F}_{\text{sig}}$. If the checks pass, then updates values of $\text{lw}_{jk} = fw_{jk}$, $\text{lw}_{kj} = fw_{kj}$, σ_j, and σ_k for tuple $(VK_j, VK_k, \text{lw}_{jk}, \text{lw}_{kj}, \sigma_j, \sigma_k)$ in nTable. $\mathcal{F}_{\text{pay}}$ finally returns "success" to node_j and node_k.

Figure 9: Ideal functionality for pay phase

For each non-RN node (node_i) in the network, on receiving $\textit{hold}_{(\cdot)}$ message, $\mathcal{S}$ calls $\textbf{verifyandCreateNextHold}_{(\cdot)}$ with the received message. $\mathcal{F}_{\text{hold}}$ verifies the contents and updates the message as needed and replies back with the updated message and the neighbor node_j the message should be forwarded to. $\mathcal{S}$ then forwards the *hold* message to node_j and also calls multisig on behalf of node_j and node_i to create multisig hold contracts between them. If the non-RN node is controlled by $\mathcal{A}$, and does not respond, the simulation times out. If it responds, but with an incorrect reply, it will be caught at the point when the reply passes onto a honest node.

For each RN node (node_i) in the network, on receiving *hold*$_{(.)}$ message msg and if node_i is a RN, $\mathcal{S}$ sends
$(\textbf{verifyandcreatenext}\mathrm{RN}\textit{hold}_{(.)},\ msg)$ to $\mathcal{F}_{\text{hold}}$. For RN_s and all RNs between RN_s and RN_r, $\mathcal{F}_{\text{hold}}$ internally verifies the messages, updates the *hold*$_s$ message contents, and returns the message to be forwarded by each RN along the path. If node_i is RN_r, the $\mathcal{F}_{\text{hold}}$ stores the information from the *hold*$_s$ and *hold*$_r$ tuple internally and when both messages are received, replies to $\mathcal{S}$ with a signed *proceedPay* tuple on behalf of RN_r that the $\mathcal{S}$ can forward to receiver of transaction. $\mathcal{F}_{\text{hold}}$ also sends a *holdACK*$_{(.)}$ message to $\mathcal{S}$ after processing a *hold* message on behalf of an RN. If RN_s is controlled by $\mathcal{A}$, it might try to find a longer path or not send a *holdACK* message back to the sender. Our threat model does not consider finding a longer path as malicious behaviour, as long as the amount is reserved and decremented correctly. If RN_s mis-routes or returns wrong/garbage information, e.g., wrong $\mathrm{hopCount}$, ν, etc. $\mathcal{F}_{\text{hold}}$ will return $\perp$ inside **verifyandcreatenextRNHold**. The case where RN_r or any of the intermediate RNs are malicious is handled similarly.

If $\mathcal{F}_{\text{hold}}$ returns $\perp$ for $\textbf{verifyandcreatenext}\textit{hold}_{(.)}$ or $\textbf{verifyandcreatenext}\mathrm{RN}\textit{hold}_{(.)}$ then $\mathcal{S}$ sends $\mathrm{createHoldRej}$ message to $\mathcal{F}_{\text{hold}}$ and $\mathcal{A}$ if needed. $\mathcal{F}_{\text{hold}}$ returns the *holdReject*$_{(.)}$ tuple to $\mathcal{S}$ which is sent by node_i along with $\mathrm{multisigRev}$ function to the neighbor it received *hold*$_{(.)}$ message from. Each node in the network calls $\mathrm{verifyHoldAck}$ and $\mathrm{verifyHoldReject}$ on receiving *holdACK*$_{(.)}$ or *holdReject*$_{(.)}$ message respectively. If verification passes in the real world for *holdACK* or *holdReject*, the node forwards the message towards next neighbor who had sent the *hold* message. In case of *holdReject* message, the node additionally calls $\mathrm{multisigRev}$ function with the neighbor it sent *holdReject* message to.

On receiving *proceedPay* tuple, $\mathcal{S}$ sends $\mathrm{createPay}$ tuple to $\mathcal{F}_{\text{pay}}$ on behalf of the receiver. There is also the possibility that RN_r is controlled by $\mathcal{A}$ and RN_r never responds, in which case the simulation aborts, or RN_r responds with malformed tuples or messages which is handled by the $\mathrm{verifyPay}$ interface as described below. $\mathcal{F}_{\text{pay}}$ responds with a *pay* tuple. $\mathcal{S}$ can send the *pay* tuple to next hop neighbor of the receiver along the transaction path. Each honest node along the path on receiving the *pay* tuple would have $\mathcal{S}$ verify the received tuple by calling $\mathrm{verifyPay}$ function provided by $\mathcal{F}_{\text{pay}}$ and if the verification passes, then $\mathcal{S}$ on behalf of each node pair calls $\mathrm{multisigPay}$ function to update the link weights between the nodes. Whenever, RN controlled by $\mathcal{S}$ receives the *pay* tuple, $\mathcal{S}$ calls $\mathrm{verifyPayRN}$ on behalf of the node and $\mathcal{F}_{\text{pay}}$ verifies the message and returns a *payACK* tuple if verification passes. $\mathcal{S}$ then sends the *payACK* tuple to the node it received *pay* tuple from before forwarding the *pay* tuple to the next node along the transaction path. Each node on receiving the *payACK* tuple calls $\mathcal{F}_{\text{pay}}$ function $(\mathrm{verifyPayAck}$ through $\mathcal{S}$ to verify the *payACK* message and if verification passes then the transaction timers are cleared. When the sender finally receives *pay* tuple, $\mathcal{S}$ calls $\mathrm{verifyPaySender}$ function of $\mathcal{F}_{\text{pay}}$ on behalf of the sender. $\mathcal{F}_{\text{pay}}$ verifies the *pay* message and returns a *payACK* message to $\mathcal{S}$ which is then forwarded and verified among all nodes between the sender and RN_s. When RN_s receives and verifies the *payACK* message originated at the sender, the transaction is considered complete. □