Auroch: Auction-Based Multipath Routing for Payment Channel Networks

Mohammed Ababneh
New Mexico State University
Las Cruces, NM, USA
mababneh@nmsu.edu

Kartick Kolachala
New Mexico State University
Las Cruces, NM, USA
kart1712@nmsu.edu

Roopa Vishwanathan
New Mexico State University
Las Cruces, NM, USA
roopav@nmsu.edu

ABSTRACT

The Bitcoin blockchain scalability problem has inspired several off-chain solutions for enabling cryptocurrency transactions, of which Layer-2 systems such as payment channel networks (PCNs) have emerged as a frontrunner. PCNs allow for path-based transactions between users without the need to access the blockchain. These path-based transactions are possible only if a suitable path exists from the sender of a payment to the receiver. In this paper, we propose *Auroch*, a distributed auction-based pathfinding and routing protocol that takes into account the routing fees charged by nodes along a path. Unlike other routing protocols proposed for PCNs, *Auroch* takes routing fees into consideration. *Auroch* maximizes the profit that can be achieved by an intermediate node at the same time minimizing the overall payment cost for the sender.

CCS CONCEPTS

• **Security and privacy → Privacy-preserving protocols**; **Distributed systems security**; **Security protocols**.

KEYWORDS

Payment channel networks, Auction, Blockchains.

ACM Reference Format:
Mohammed Ababneh, Kartick Kolachala, and Roopa Vishwanathan. 2024. *Auroch*: Auction-Based Multipath Routing for Payment Channel Networks. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '24), July 1–5, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3634737.3657021

1 INTRODUCTION

Blockchains enabling cryptocurrencies such as Bitcoin have inherent scalability problems, e.g., Bitcoin supports less than 7 transactions per second [19] and Ethereum supports less than 15 transactions per second [27], as compared to traditional financial systems, e.g., Visa processes over 24,000 transactions per second [32]. Payment channel networks (PCNs) provide a solution to the scalability challenges faced by blockchains, by enabling users to take part in transactions without publishing each transaction to the blockchain. Examples of PCNs include Lightning Network [19], Raiden [21], and Ripple [22]. A *payment channel* is essentially a single multisignature blockchain transaction between two parties that locks up coins for a fixed amount of time for use only between those two parties. In contrast to the on-chain transaction confirmation mechanism used in blockchains, once a payment channel has been established between two parties, all transactions between them utilizing the locked-up coins can be performed offline without the need to write transactions on the blockchain. The available balance, i.e., coins of a channel limits the number of transactions that can be performed over that channel. Two nodes that do not have a direct payment channel can still transact with each other using multiple transactions over payment channels between intermediate nodes, forming a *payment channel network*. To facilitate transactions between a sender and a receiver, intermediate nodes charge routing fees. As a result, the sender node has to pay both the actual amount that they intend to send, along with the sum of routing fees along the path that comprises several intermediate nodes. Figure 1 represents a PCN with six nodes Alice, Sam, Alexa, Peter, Megan, and Chloe. The bidirectional edges between the nodes represent the payment channels which allow funds to be sent in both directions, the edge weights next to each node depict that node's local balance, and the sum of the nodes' local balances represents the channel capacity.

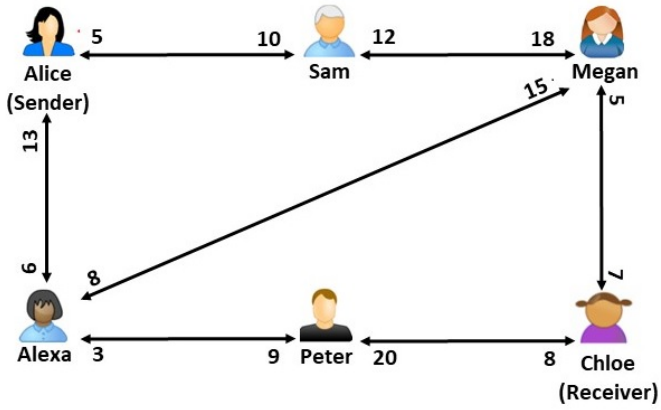


Figure 1: Payment channel network

If Alice wishes to send a payment to Chloe, she will need to route the payment through intermediaries, since there is no direct Alice–Chloe link. Alice can forward the payment along the Alice → Sam → Megan → Chloe path or the Alice → Alexa → Peter → Chloe path, or any of the other paths. Since Alexa, Sam, Peter, and Megan are forwarding Alice's payment, they will charge a certain amount for their service, typically called *routing fees*. Since nodes in a PCN are not assumed to know the entire network topology beyond their immediate neighborhood, one of the big challenges in

ASIA CCS '24, July 1–5, 2024, Singapore, Singapore

ACM ISBN 979-8-4007-0482-6/24/07.
https://doi.org/10.1145/3634737.3657021

PCNs is finding a path from a sender to a receiver in a decentralized, privacy-preserving way. In a PCN, pathfinding depends on several factors. One factor is the capacity or available liquidity of any chosen path from a sender to a receiver should at least be equal to the intended payment. In turn, a path's capacity is upper bounded by the minimum capacity of the payment channels between the path's intermediate nodes. In Figure 1, consider the path Alice → Alexa → Peter → Chloe. If Alice intends to send a payment via this path, the maximum amount she can send is three coins. A second important factor, that is of interest in this work, is the routing fees charged by intermediate nodes for routing payments. This is especially important since prior works [13, 17, 25, 26, 29] do not consider routing fees. Though there are prior works in the literature that consider routing fees [4, 7, 37, 38], they either use centralized routing techniques for pathfinding or do not have mechanisms in place that maximize the profit of intermediate nodes (nodes along the path from the sender to receiver, excluding the sender and receiver). *Auroch* differs from these since in *Auroch*, we do not use a centralized routing mechanism for pathfinding and we also maximize the overall profit of the intermediate nodes. We envision a scenario in which a path is constructed in a backward direction from a receiver to a sender, and each node is offered the option of joining a transaction path by a neighboring node that has already joined the path. In Figure 1, from the perspective of an intermediate node Megan, it is beneficial to select the best node among her neighbors Sam or Alexa in terms of higher routing fees and channel capacity. Similarly, an intermediate node Alexa might be offered to join two paths Megan → Chloe or Peter → Chloe, then from Alexa's perspective, it is desirable to join the path that offers the largest capacity and lowest cost. In summary, including routing fees into distributed pathfinding protocols provides an incentive for intermediate nodes to participate in transactions. This has the effect of increasing the number of paths from sender to receiver. On the other hand, the sender's goal is to utilize the set of available paths, with their different capacities and costs, to minimize the overall cost of the payment. To provide a solution to these conflicting goals, this paper makes the following contributions:

(1) We develop a distributed pathfinding and routing protocol, *Auroch*, that provides incentives for intermediate nodes to collaborate in route formation.
(2) To maximize the intermediate nodes' profit, while minimizing the routing cost for the sender, we provide a linear programming-based formula to divide the payment into smaller amounts, each of which can be forwarded along different paths.
(3) We rigorously analyze and prove the security of *Auroch* in the Universal Composability framework.
(4) The performance of *Auroch* is evaluated on real-world PCN datasets, and shows that it outperforms other comparable routing protocols for PCNs, in terms of success ratio, volume of successful transactions, and routing fees.

Auroch can potentially be leveraged to other autonomous decentralized networks with unpredictable connectivity and high turnover rates, such as IoT or edge networks.

Outline: In Section 2, we discuss related work. In Section 3, we describe relevant concepts in auction theory. In Section 4 we give a brief overview of *Auroch* and in Section 5, we present its construction. In Section 6, we present our experimental results, in Section 7 we analyze the security of *Auroch* in the UC framework, and in Section 8, we conclude the paper.

2 RELATED WORK

Routing protocols which do not consider routing fees: Flare [20] adopts beacon-based routing, in which each node stores local information about the network and also stores information about *beacon* nodes, which are highly connected (nodes with a large number of incoming and outgoing links). A sender initially tries to find a path to the receiver using her locally stored information, if she is unable to do so, she contacts the nearest beacon node to facilitate pathfinding. Flare uses source routing in which each node is expected to store data about the network topology and it also does not take into account the routing fees charged by nodes. SilentWhispers [13] leverages the concept of landmark routing [31], in which a well-connected node in the network called *landmark* aids the other nodes in the pathfinding process. The sender first finds a path from itself to the landmark and the landmark then finds a path from itself to the receiver. Both these paths are stitched together to get the end-to-end path from the sender to the receiver. This protocol requires every path from the sender to the receiver to pass through a landmark, even though a shorter path is available that does not involve a landmark and does not take the routing fees charged by nodes into consideration during path selection. SpeedyMurmurs [26] is a routing protocol that uses prefix embedding for pathfinding. The network is organized as a tree, every node is assigned a prefix and the children of a node derive their prefixes from the parent's prefix. This protocol calculates the path between two nodes based on the length of the common prefix between them. However, like [13], it does not take into account the routing fees charged by the intermediate nodes. Blanc [17] is a pathfinding protocol that uses a subset of users to facilitate transactions, termed "Routing Helpers". Pathfinding from sender to receiver uses the technique of broadcast flooding, due to which the pathfinding phase incurs a very high communication overhead. Apart from this, it also uses the blockchain as an auditing mechanism. This results in a significantly high number of blockchain writes, even for successful transactions; in the event of transaction retries, the overhead is a lot more. This protocol too like the prior ones described so far, does not take into account routing fees charged by an intermediate node during routing. Spider [29] is a pathfinding protocol that uses source routing, in which the sender divides the payment into several smaller payments, each of which will be sent through a different path. This protocol mainly focuses on avoiding depletion of nodes' link balances avoiding skewness (a situation where the node's link weights become depleted) in the network and stores transactions in queues (each node maintains its queue). This causes the transactions to wait for an indefinite amount of time before they are processed. It also does not take into account the privacy of nodes involved in the transaction.

Routing protocols that incorporate routing fees: The distributed protocols Cheapay[38] and Robustpay+[37] both use source routing and a flat fee structure, but they differ in the aspect that Cheapay sends a single payment on a single path, whereas Robustpay+ uses

Table 1: Comparison of routing protocols that factor in routing fees

Routing Protocols	Concurrency	Sender/Receiver Privacy	DSR/CR	SR	Fees	Payment Structure		MIP	DCB
						Path	Payment		
Auto-tune [8]	No	Yes	DSR	Yes	Flat	Multiple	Multiple	No	No
MPCN-RP [4]	No	No	CR	Yes	Flat	Single	Single	No	No
VEIN [7]	No	No	CR	Yes	Dynamic	Multiple	Multiple	No	No
Robustpay+ [37]	No	Yes	DSR	Yes	Flat	Multiple	Redundant	No	No
Cheapay [38]	No	Yes	DSR	Yes	Flat	Single	Single	No	No
Auroch	Yes	Yes	DSR	No	Dynamic	Multiple	Multiple	Yes	Yes

redundant payments on multiple paths. Both MPCN-RP[4] and VEIN[7] use centralized techniques in conjunction with source routing; however, VEIN uses dynamic fees and multiple payments on multiple paths. MPCN-RP uses flat fees and single payment on a single path. Auto-tune [8], similar to Cheapay and Robustpay+ uses source routing and implements a flat fee structure. With support for concurrency, sender/receiver privacy, distributed routing without source routing, dynamic fees, single payments, intermediaries' profit maximization, and dynamic channel balance adjustments, *Auroch* maintains its distinction from other works published in the domain. In Table 1, we give a qualitative comparison between *Auroch* and other routing protocols that consider routing fees. We now briefly describe our comparison metrics. Concurrency indicates whether the routing protocol allows the node to route multiple transactions at the same time. Send/Rec privacy indicates that the sender/receiver identity should not be known by other nodes in the network. Distributed/Centralized Routing (DSR/CR) indicates whether the routing protocol employs a trusted node (e.g., landmark, coordinator, etc.) in the network to provide routing services. Source routing (SR) indicates whether the sender constructs the end-to-end path to the receiver. The intermediate nodes can either charge a fixed fee called a flat fee or a dynamic fee based on factors such as the transaction amount. The payment structure indicates how the payment is routed. A single payment is when the transaction amount is not split by the sender. Multiple payments refer to when the amount is split into multiple chunks by the sender. A redundant payment is the same amount being transacted across several paths where the payment that reaches the receiver first is recorded. A single path is the sender using only one path to route payments to the receiver. Multiple paths are the sender using more than one path. Maximization of intermediaries profit (MIP) indicates whether the routing protocol aims to maximize the monetary profit of the intermediate nodes. Dynamic channel balance (DCB) indicates whether the routing protocol considers the dynamic changes in the channel balance of nodes during path construction.

Auction Theory: The way buyers and sellers exchange goods in a market is the subject of a sub-discipline of economics known as auction theory [18]. We now give the definitions of terms from the auction literature [28, 36] that we use in *Auroch*. **Bidder** is the entity that submits a request in the auction to buy or sell commodities. In *Auroch*, any node in the network can be a bidder. **Seller** is the entity that owns a commodity and wants to sell it. In *Auroch*, any node in the network can be a seller. **Auctioneer** is the entity that supervises and controls the bidding process. In *Auroch*, any node in the network can conduct its auction and thus can have unilateral control over it. **Commodity** refers to the object traded in the auction between the buyer and seller. In *Auroch*, the commodity is the path capacity between a pair of nodes. **Valuation** is a metric used by the nodes to compare and decide which path needs to be chosen in the event of having more than one path to route the payment. The higher the valuation price for a path, the more likely it is for that path to be chosen to route the payment. **Blockchain**: *Auroch* can be deployed on any blockchain that supports hashed time-locked contracts (HTLCs). Wireless networks deploy auctions to create an appropriate reward that will encourage relay nodes to forward data [35]. In wireless networks, energy, processing power, and bandwidth are the resources that are auctioned. On the other hand, in PCNs, coins on the payment channel are the resources that are auctioned. Owing to the very different features and intent of both networks, auctions from wireless networks cannot be trivially ported to PCNs.

There are various auctioning techniques available, including first price, second price auctions [36], and proportional auctions [1]. We do not use first-price or second-price auctions since it is not clear whether one can deploy them in a multipath payment scenario; rather in *Auroch*, we use proportional auctions [1] which seem a natural fit since they are commonly used in contexts where the commodity being auctioned is divisible, such as path capacity in our case. In *Auroch*, bidders will compete for a path capacity that is being offered for sale. Each bidder makes a bid in a proportional auction, indicating the portion of the path capacity they are willing to buy and the price they are prepared to pay per unit of that path capacity. The winners in this auction are determined by a linear optimization formula that takes into account both of the above factors.

3 SYSTEM SETUP

Path Capacity: Path capacity is the defined as a the minimum amount that can be transacted between a sender and a receiver along a given path comprising of several intermediate nodes. In the Figure 2a, the path capacity along the path Alice → Peter → Ivan → Chloe is 15.

Channel Capacity: Channel capacity is defined as the total amount (balance) two nodes have in their payment channels with each other. In Figure 2a, the channel capacity between Alice and Peter is (25+50 = 75).

Network model: A payment channel network (PCN) can be modeled as a directed graph $G = (V, E)$ in which the set of vertices V represent network nodes (i.e., users) and E represents the set of

directed edges (i.e., payment channels) between nodes. Assume nodes v_i and v_j have established a payment channel $e_{i,j}$ between them. Then, let $b_{i,j}$ represent the balance from node i to j (i.e., how many coins can node i forward in the direction of node j). Note that the balances $b_{i,j}$ and $b_{j,i}$ are not necessarily equal. The channel capacity of a payment channel channel $e_{i,j}$ is denoted as $c_{i,j}$, where $c_{i,j} = b_{i,j} + b_{j,i}$.

Challenges: One of the main challenges in applying auction mechanisms to PCN routing is the quantification of one path's efficiency in comparison to other possible paths. Path efficiency is a function of several factors such as the fee the path requires, the number of hops, and the path capacity it can provide. To overcome this challenge, *Auroch* proposes a new valuation price function to evaluate the paths. The second challenge is the design of the bidding price mechanism with the restriction of keeping the intended value of the transaction secret (from all intermediaries) until forwarding paths are selected. In *Auroch*, the bidding price mechanism is based on the maximum capacity of intermediaries' channels and not on the transaction amount. Third, the local channel balances of nodes in the payment channel network should not be revealed to nodes other than immediate neighbors, since doing so is a privacy violation. To overcome this challenge, *Auroch* enables the nodes to proactively apportion funds to different transactions, thus enabling concurrency.

Threat model and Assumptions: In *Auroch*, we assume that every node in the PCN has access to the partial view of the network topology and no node in the PCN knows the entire network topology. This assumption has been made since making the entire network topology public will reveal the identities of all the nodes violating their privacy. One of the major advantages of blockchain based financial transactions is the ability to execute them anonymously. Apart from this, every node maintaining the entire network topology locally incurs a tremendous storage overhead and it may not be possible for nodes that operate the PCN from resource constrained devices. The Lightning Network, one of the most popular PCNs has implemented trampoline routing protocol [30] which requires nodes only to maintain a partial view of the network topology.

The sender, n_s and receiver, n_d in *Auroch* can be malicious. Both of them can choose to abandon a transaction or cause intentional delays in the payments to lock-up collateral in channels. The intermediate nodes in *Auroch* can be un-trusted and can act in arbitrary ways. These nodes can misreport the maximum capacity they have in their channels with their next-hop neighbors with the intention of causing transaction failures. In this paper, we assume that there is at least one viable path available for the n_s to route her payment. We assume that all users have a long-term verification/signing key-pair and all users have pseudonymous, temporary key-pairs. Users' temporary key-pairs do not change unless there is a dispute.

Security/privacy properties: *Balance security*: Balance security in our system ensures that any honest user participating in a transaction does not lose coins even if all other users engaging in the transaction are corrupted. *Sender privacy*: In our system, sender/receiver privacy is realized when an adversary cannot identify the identity of the sender in a transaction between honest users. *Link privacy*: An adversary can determine the path capacity along a path comprising of several intermediate nodes but can never exactly the local balance an honest node's payment channel with its honest immediate neighbor. *Value privacy*: The transacted value is not learned by an adversary who is not involved in the payment path between sender and receiver. *Bidding values privacy*: Bidding value is typically sensitive information in an auction, since it reflects the economic strength of the buyer. Therefore, all bidding values should be protected against rival bidders.

4 OVERVIEW OF *Auroch*

The goal of this paper is to explore the idea of using an auction mechanism to find multiple paths between a sender/receiver pair in a PCN and splitting a transaction amount along the different paths. We aim to do this in a way such that intermediaries can choose the most profitable path(s) to be on (in terms of routing fees), which incentivizes them to participate in transactions while helping the sender choose the most inexpensive path to the receiver. We use principles from auction theory and linear optimization to help achieve our goal.

4.1 *Auroch* Stages

Auroch consists of three main stages: (1) Setup and Route discovery, (2) Route auctioning, and (3) Route selection. Next, the steps of the different stages are explained in detail.

Setup and Route Discovery: In a dynamic environment such as a PCN, there is a possibility of the payment channel closing or opening, the node becoming offline, and frequent changes in the payment channel balance. Route discovery using route request (RREQ) and route reply (RREP) messages form the foundation of the Aourch. Consequently, a node should start a fresh route discovery procedure whenever it wishes to send a payment while the destination's route is unknown. To find a path to the intended destination, the node broadcasts a route request (RREQ) message. Without loss of generality, let n_s denote a node that wants to make a payment amt to a destination node n_d. To initiate pathfinding to the destination, node n_s broadcasts a routing request RREQ to its neighboring nodes (i.e., nodes connected with a payment channel to node n_s). For our purposes in this work, the RREQ message contains a field as follows RREQ = [txid]. The txid is the unique ID of the request. Each node that receives an RREQ tuple locally stores the txid and forwards this tuple to all its immediate neighbors. This step is repeated until the RREQ tuple reaches the intended destination, n_d. If a node in the PCN does not want to be involved in the transaction, it simply drops the RREQ tuple.

Route Auctioning: Once a destination node n_d is reached, it sends RREP messages to the nodes in the set $Parent(n_d)$ that n_d maintains locally. This set contains the immediate neighbors of n_d from whom n_d has received the RREQ tuple. The reply message RREP consists of a number of fields as follows RREP = [txid, $P(n_d, n_l)$, $Cap(n_d, n_l)$, $Cost(n_d, n_l)$]. The field $P(n_d, n_l)$ denotes the node path from the destination n_d down to the current node n_l. The field value $Cap(n_d, n_l)$ indicates the path capacity that can be allocated along the path $P(n_d, n_l)$ and $Cost(n_d, n_l)$ refers to the corresponding forwarding cost per token. Once n_l receives the RREP message, it conducts an auction to allocate both path capacity and cost, which are running values that will be updated at each hop. The route auctioning process consists of the following phases (i.e., sub-phases);

(1) **Auction setup :** Let n_a be an intermediate node (auctioneer) along the path from n_s to n_d which maintains a set, $Bidder(n_a)$, which contains the nodes from whom n_a has received an RREQ message. Node n_a then auctions the path capacity along the path from node n_d to node n_a. To start the auction, node n_a broadcasts the AUC tuple to all the nodes in the set $Bidder(n_a)$. This message contains the following fields AUC = [txid, $MaxCap(n_d, n_a)$, $ResPrice(n_d, n_a)$]. In particular, $MaxCap(n_d, n_a)$ indicates the path capacity that node n_a can provide over the path $P(n_d, n_a)$ (which is upper bounded by the local channel balance of n_a with its immediate neighbor along the path from n_d to n_a). The third field in the AUC message is the reservation price $ResPrice(n_d, n_a)$ indicates the minimum price that the auctioneer n_a demands to forward a payment along the path. Assume that node n_a has won the preceding auction for the path $P(n_d, n_{a-1})$ and thus has attained forwarding privilege for the path $P(n_d, n_a)$. Let $WinPrice(n_a, P(n_d, n_a))$ denote the price that n_a committed to pay to n_{a-1}. Then, the reservation price $ResPrice(n_d, n_a)$ that node n_a asks for can be given as follows

$$ResPrice(n_d, n_a) = WinPrice(n_a, P(n_d, n_a)) + \tau_{\text{fee}} \tag{1}$$

where, $\tau_{\text{fee}} \in [f_{\min}, f_{\max}]$ denotes the fee that node n_a charges as its forwarding fee. After nodes in $Bidder(n_a)$ receive an AUC message from n_a, the bidding process is initiated.

(2) **Bidding Setup :** In this phase, after receiving an AUC message, say a node $n_b \in Bidder(n_a)$ offers bidding prices for the path from n_d to n_a. Note that node n_b might receive several AUC messages from several auctioneer nodes, M, which are the nodes to whom n_b has sent an RREQ message. These auctioneer nodes are added to the set $AN(n_b) = AN(n_b, 1), AN(n_b, 2), \ldots, AN(n_b, M)$, that is maintained locally by the node n_b. For our purposes in this work, BID = [txid, $BidCap(n_b, P(n_d, m))$, $BidPrice(AN(n_b, m))$].
In particular, $BidCap(n_b, P(n_d, m))$, where $m \in [1..M]$, indicates the maximum amount that node n_b is willing to send along the path $P(n_d, m)$. The third field in the BID message is $BidPrice(AN(n_b, m))$ that denotes the bidding price that n_b offers to each auctioneer node m. We define two pricing rules for *Auroch* to determine the $BidPrice$.

- **Pricing Rule 1**: To be able to participate in a bid, the bidding price should at least satisfy the reservation price set by the auctioneer. Given the forwarding fee range interval $[f_{\min}, f_{\max}]$, a simple price rule for each auctioneer node m can be proposed as follows:

$$BidPrice(AN(n_b, m)) = ResPrice(n_d, n_a) + \tau_1 \tag{2}$$

where, $\tau_1 \xleftarrow{\$} [f_{\min}, f_{\max}]$. [1]

- **Pricing Rule 2**: Pricing rule 1 did not take all information into account for bid price determination. For example, neither the number of auctioneers nor their offered capacities are incorporated. To account for these parameters, we define a normalized valuation price $VtPrice(n_b, AN(n_b, m))$ computed as follows:

$$VtPrice(n_b, AN(n_b, m)) = \frac{MaxCap(n_d, AN(n_b, m))}{\sum_k MaxCap(n_d, AN(n_b, k))} \times \left[1 - \frac{ResPrice(n_d, AN(n_b, m))}{\sum_k ResPrice(n_d, AN(n_b, k))}\right] \tag{3}$$

$VtPrice(n_b, AN(n_b, m))$ value falls in the interval $[0, 1]$. This equation helps the bidding node to differentiate between the reservation prices and offered capacities of the M auctioneers' offers.
In the first term of Equation 3, the capacity of an auctioneer is compared relative to the sum of all offered capacities (i.e., normalized to a maximum value of 1). The more capacity an auctioneer offers relative to other auctioneers, the more is the value associated with this capacity and consequently, it is better to use the path offered by that auctioneer node. In the second term of Equation 3, the reservation prices are compared. Similar to the first term, an auctioneer's reservation price is compared to the sum of all prices offered by all other auctioneers to normalize it for comparison purposes.
Thus, the normalized reservation price indicates how costly a certain auctioneer node is in comparison to other auctioneers. In contrast to the capacity parameter, the larger the reservation price requested by a auctioneer in comparison to other auctioneers, the worse it is to use the path offered by this auctioneer since it incurs a higher fee value to be paid by the bidder. Thus, to compare the capacity and the reservation price together, the normalized reservation price is subtracted from 1. Finally, we note that since both the capacity and reservation prices are normalized to 1, their product (the proposed valuation price) is also normalized. Hence, an auctioneer node with greater offered capacity and lower reservation price relative to other auctioneers has a larger $VtPrice(n_b, AN(n_b, m))$.
Using the $VtPrice(n_b, AN(n_b, m))$, value, we propose the following bidding rule

$$BidPrice(AN(n_b, m)) = ResPrice(n_d, n_a) + \tau_2 \tag{4}$$

where τ_2 is given as follows

$$\tau_2 = f_{\min} + VtPrice(n_b, AN(n_b, m))(f_{\max} - f_{\min}) \tag{5}$$

Using this rule, we note that the higher the capacity and the lower the cost an auctioneer node offers, the more bidding price the bidder is willing to pay for the corresponding path. If a bidding node receives a single AUC tuple, pricing rule 1 is used. If a bidding node receives multiple AUC tuples, it could either use pricing rule 1 or pricing rule 2, but it is more optimal for the bidding node to use pricing rule 2.

(3) **Auction results and announcing:** Upon receiving BID messages, the auctioneer node n_a must determine how to allocate the path capacity along the path $P(n_d, n_a)$ among bidding nodes. In particular, given a set of bidding node prices and the maximum bid capacities, The auctioneer node n_a should allocate the path capacity along the path among

[1] The values of f_{min} and f_{max} are 1, 10 respectively for Lightning Network [34].

bidding nodes to maximize the total forwarding cost. it is important to note that node n_a may receive BID messages from a number of B nodes and these nodes are added to set $BN(n_b) = BN(1, n_a), BN(2, n_a), \ldots, BN(B, n_a)$, which is locally maintained locally by node n_a.
The described problem of path capacity allocation can be formulated as a linear optimization problem :

$$\max_{Cap(n_d,BN(b,n_a))} \sum_b \text{BidPrice}(AN(b, n_a))\text{Cap}(n_d, BN(b, n_a)) \quad (6a)$$

$$\text{s.t.} \quad \sum_b \text{Cap}(n_d, BN(b, n_a)) \leq \text{Cap}(n_d, n_a) \quad \forall b \quad (6b)$$

$$\text{Cap}(n_d, BN(b, n_a)) \leq \text{BidCap}(BN(b, n_a), P(n_d, n_a)), \quad \forall b \quad (6c)$$

$$\text{Cap}(n_d, BN(b, n_a)) \geq 0 \quad \forall b \quad (6d)$$

The above problem is a linear optimization problem since the cost function as well as the constraints are linear in $Cap(n_d, BN(b, n_a))$. The constraint in Equation 6b ensures that the sum of maximum capacities of the bidders is less than or equal to path capacity along the $P(n_d, n_a)$. Hence, this serves as an upper bound on the sum of the maximum capacities of the bidders. The constraint in Equation 6c, guarantees that the assigned path capacity $Cap(n_d, BN(B, n_a))$ does not exceed the maximum amount that bidder is willing to send along the the path $P(n_d, n_a)$.The last constraint (Equation 6d) enforces the non-negativity of the assigned maximum capacity. An auctioneer node n_a determines to which node(s) the available path and path capacity should be allocated. To advertise the auction results, the auctioneer node sends the *RREP* message to the winning nodes. As stated earlier, the *RREP* consists of the following fields $RREP = [\text{txid}, P(n_d, n_l), Cap(n_d, n_l), Cost(n_d, n_l)]$. The process of message replies and performing auctions is repeated until the source node n_s becomes a bidding node and bids for the different paths from the auctioneer nodes.

Route Selection: In this stage, node n_s is to determine which of the available paths to use for routing payments. In particular, given the set of path costs and capacities, n_s needs to send partial payments over these paths such that the total forwarding cost it has to pay is minimized. The payment allocation problem can be formulated as a linear optimization problem, which we describe below.

Let $\mathcal{P}(n_d, n_s) = [P(n_d, n_s)^1, P(n_d, n_s)^2, \ldots, P(n_d, n_s)^K]$ denote the set of K paths between n_s and n_d. Furthermore, let $\mathcal{C}(n_d, n_s) = [Cap(n_d, n_s)^1, Cap(n_d, n_s)^2, \ldots, Cap(n_d, n_s)^K]$ and $\mathcal{F}(n_d, n_s) = [Cost(n_d, n_s)^1, Cost(n_d, n_s)^2, \ldots, Cost(n_d, n_s)^K]$ denote the sets of path capacities and costs, respectively. Let $p^k_{(n_s,n_d)}$ denote the partial payment over the k^{th} path, where $k \in [1..K]$, and $m \in [1..M]$. The payment allocation problem can be stated as follows:

$$\min_{p^k_{(n_s,n_d)}} \quad \sum_m Cost(n_d, n_s)^k p^k_{(n_s,n_d)} \quad (7a)$$

$$\text{s.t.} \quad \sum_k p^k_{(n_s,n_d)} \geq amt \quad (7b)$$

$$\sum_k p^k_{(n_s,n_d)} \leq amt + \sum_k Cap(n_d, n_s)^k Cost(n_d, n_s)^k \quad (7c)$$

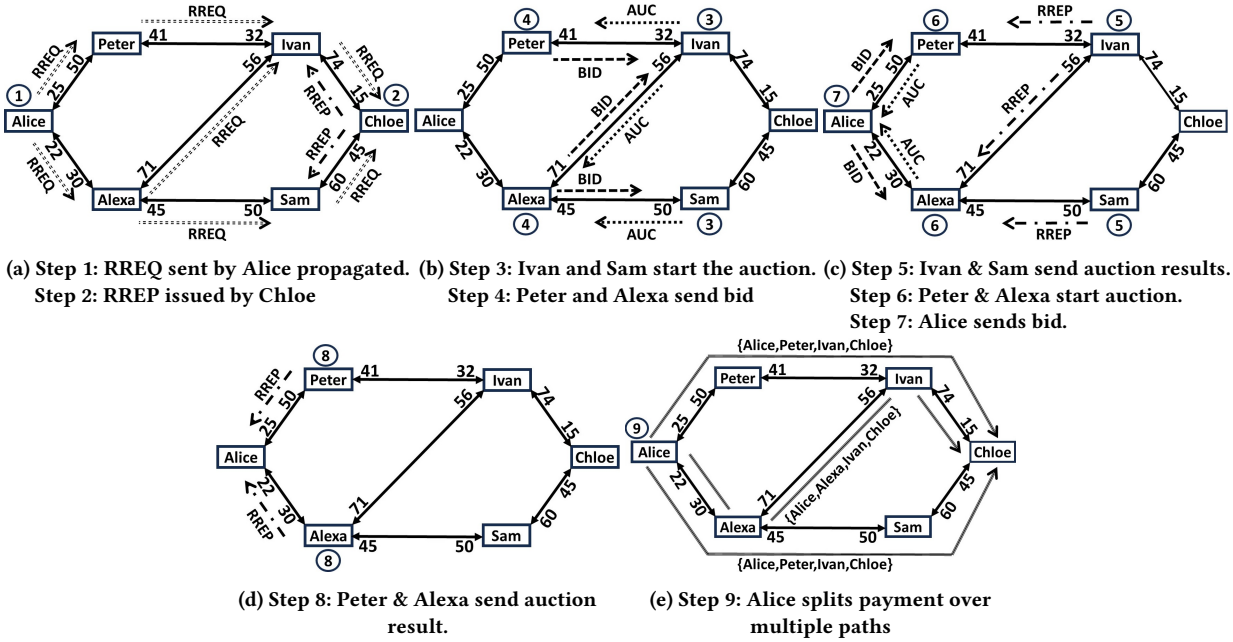
$$p^k_{(n_s,n_d)} \leq Cap(n_d, n_s)^k \quad \forall k \quad (7d)$$

$$p^k_{(n_s,n_d)} \geq 0, \quad \forall k \quad (7e)$$

Table 2: Notations Used

Notation	Description
λ	Security parameter
n	Number of nodes in the PCN
$(\text{sk}_i, \text{vk}_i)$	Node i's long-term signing, verification key-pair
$(\text{SK}_i, \text{VK}_i)$	Node i's temporary signing, verification key-pair
n_s	Sender Node
$\mathbb{I}_{n_s}$	Set of immediate neighbors of a n_s
n_d	Destination Node
n_l	Current node
n_a	Auctioneer node
n_b	Bidder node
RREQ	Route request message
RREP	Route replay message
txid	Transaction ID
$P(n_d, n_s)$	Path from destination to sender
$P(n_d, n_l)$	Path from destination to current node along path $P(n_d, n_s)$
f_{min}, f_{max}	Minimum and maximum routing fees that can be charged by a node
$\mathcal{P}$	a list of available paths from n_s to n_d
$\mathcal{C}$	a list of available paths capacities from n_s to n_d
$\mathcal{F}$	a list of available paths costs from n_s to n_d
$Bidder(n_a)$	a set of bidder nodes for node n_a
$AN(n_b)$	a set of verified auctioneer nodes for bidder n_b
$BN(n_a)$	a set of verified bidding nodes for auctioneer n_a

The above problem is a linear optimization problem since the cost function as well as the constraints are linear in $p^k_{(s,d)}$. The constraint in Equation 7b ensures that the sum of partial payments is either greater than or equal to the original payment amount amt. Hence, this serves as a lower bound on the sum of payments. The payment amount on each path should take the path's cost into consideration. Thus, the total amount to be payed is upper bounded by the sum aggregate of all forwarding cost across all the paths between n_s and n_d. This is ensured by the constraint in Equation 7c. The constraint in Equation 7d, guarantees that a



(a) Step 1: RREQ sent by Alice propagated. Step 2: RREP issued by Chloe

(b) Step 3: Ivan and Sam start the auction. Step 4: Peter and Alexa send bid

(c) Step 5: Ivan & Sam send auction results. Step 6: Peter & Alexa start auction. Step 7: Alice sends bid.

(d) Step 8: Peter & Alexa send auction result.

(e) Step 9: Alice splits payment over multiple paths

Figure 2: The step-by-step process of the auction mechanism in *Auroch* (Sender: Alice, Receiver: Chloe)

partial payment over each path does not exceed the path's capacity. The last constraint (Equation 7e) checks if the value of each partial payment is non-negative.

4.2 *Auroch* Workflow

As an example, consider a simplified network as shown in Figure 2, where a sender node Alice intends to pay a payment of amt to a receiver node Chloe.

In Figure 2c, **Step 1 :** Alice constructs the RREQ=[123] and sends it to her neighbors, Peter and Alexa. The RREQ=[123] message propagates through the network until it receives the destination, Chloe. **Step 2 :** Upon receiving the RREQ=[123] messages, Chloe generates two routing reply (RREP) messages : RREP = [123, (*Chloe*, *Ivan*), 0, 0], RREP = [123, (*Chloe*, *Sam*), 0, 0]. Chloe sends them to Ivan and Sam, respectively.

In Figure 2b, **Step 3 :** Upon receiving RREP messages, Ivan and Sam initiate the auction phase. Ivan computes the Reservation Price (*ResPrice*) using Equation 1, assuming the value 2. Subsequently, Ivan sets the path capacity along the path *P*(*chloe*, *Ivan*) as 30 tokens. Ivan generates two AUC = [123, 30, 2], AUC = [123, 30, 2] messages, individually sends them to Peter and Alexa. Similarly, Sam computes the Reservation Price (*ResPrice*) using Equation 1, assuming a value of 1. Sam sets the path capacity along the path *P*(*Chloe*, *Sam*) to 40 tokens. Sam generates one AUC = [123, 40, 1] message and forwards it to Alexa. **Step 4 :** With AUC messages in hand, Peter and Alexa start bidding. Peter receives one AUC message only from Ivan and proceeds to calculate Bid Price (*BidPrice*) using Equation 2. Peter constructs BID = [123, 22, 2.8] and transmits it to Ivan. In contrast, Alexa receives two AUC messages. Utilizing Equation 4, Alexa computes Bid Price *BidPrice* . She constructs two BID messages BID = [123, 20, 2.5],BID = [123, 25, 1.5], and forwards them to Ivan and Sam, respectively.

In Figure 2c, **Step 5 :** Upon receiving BID messages, Sam receives one BID message. Subsequently, Sam constructs a RREP = [123, (*Chloe*, *Sam*, *Alexa*), 25, 1.5] message, sending it Alexa. In contrast, Ivan receives two BID messages.So, Ivan will allocate 30 tokens to Peter and Alexa by solving a linear optimization problem represented by Equation 6 to maximize his profit. Ivan constructs two RREP messages RREP = [123, (*Chloe*, *Ivan*, *Alexa*), 20, 2.5], RREP = [123, (*Chloe*, *Ivan*, *Peter*), 10, 2.3]. These messages are then sent to Alexa and Peter. **Step 6 :** Upon receiving RREP messages, Peter and Alexa start the auction. Peter computes the Reservation Price (*ResPrice*) using Equation 1, assuming a value of 2.5. Peter sets the path capacity along the path *P*(*Chloe*, *Peter*) to 10 tokens. Peter constructs AUC = [123, 10, 2.5] and sends it to Alice. MeanWhile, Alexa conducts auctions for two paths (*Chloe*, *Ivan*, *Alexa*) and (*Chloe*, *Sam*, *Alexa*). Alexa computes the Reservation Price (*ResPrice*) for each path using Equation 1. The *ResPrice* for the paths (*Chloe*, *Ivan*, *Alexa*) and (*Chloe*, *Sam*, *Alexa*) are 2.7 and 1.7, respectively. Alexa constructs two AUC messages AUC = [123, 25, 1.7], AUC = [123, 20, 2.7] and sends them to Alice. **Step 7:** Alice receives three AUC messages. Consequently, Alice computes Bid Price (*BidPrice*) using Equation 4 for the offers. Alice constructs BID message BID = [123, 10, 2.8] and sends it to Peter. Furthermore, Alice constructs two AUC messages : BID = [123, 2, 2.8], BID = [123, 25, 2] , which Alice sends to Alexa.

In Figure 2d, **Step 8:** Peter and Alexa announce the result. Peter constructs RREP = [123, (*Chloe*, *Ivan*, *Peter*, *Alice*), 10, 5.1] and forwards it to Alice. Similarly, Alexa constructs two RREP messages: RREP = [123, (*Chloe*, *Ivan*, *Alexa*, *Alice*),

2, 5.3], RREP = [123, (*Chloe*, *Sam*, *Alexa*, *Alice*), 20, 4.3], and sends them to Alice.

In Figure 2e, **Step 9:** Upon receiving RREP messages, Alice has all available paths toward the Chloe along with the maximum payments that can be accommodated over these paths and their corresponding cost. Alice can divide its payment over these routes (see Step 9 in Figure 2e) by employing the linear optimization problem as stated in Equation 7a.

5 CONSTRUCTION

Auroch consists of three stages: setup, route auctioning, and payment. For ease of reference, we give the table of notations used in Table 2.

Setup and Broadcast: This stage is described in detail in Appendix A. At a high level, this protocol handles the generation of the signing and verification key pairs of a node, and the creation and transmission of a transaction identifier, txid.

Protocol 1: Route Auctioning: Auction Setup Phase

1 n_d receives RREQ message tuple from its neighbors
2 $\forall$ nodes $n_a \in Parent(n_d)$, n_d constructs an individual tuple RREP = [txid, $P(n_d, n_a), Cap(n_d, n_a), Cost(n_d, n_a)$]
3 n_d $\text{Sign}_{\text{SK}_{n_d}}(\text{RREP}) \rightarrow \sigma_{\text{RREP}}$ and n_d sends RREP along with σ_{RREP} to each node $n_a \in Parent(n_d)$.
4 On receiving RREP message with σ_{RREP},
5 Each node n_a does **if** $\text{Verify}_{\text{VK}_{n_d}}(\text{RREP}) \rightarrow 0$ **then**
6 | Do nothing
7 **else**
8 | Each n_a computes *ResPrice* using Equation 1
9 | Each n_a construct AUC = [txid, $MaxCap(n_d, n_a), ResPrice(n_d, n_a)$] message to all nodes $n_b \in Bidder(n_a)$
10 | Each n_a does $\text{Sign}_{\text{SK}_{n_a}}(\text{AUC}) \rightarrow \sigma_{\text{AUC}}$
11 | Each n_a sends AUC message along with σ_{AUC} to each node $n_b \in Bidder(n_a)$.
/* Route Auctioning: End Auction Setup Phase */

Auction Setup Protocol 1: This protocol handles the auction setup phase in *Auroch*, once the route request RREP messages that are broadcast by n_s reach the intended n_d, n_d constructs the RREP message and sends it along with its signature (created using its temporary signing key) on the message to each auctioneer node n_a in the set $Parent(n_d)$ from whom it has received an RREQ message (Lines 1-5). Each auctioneer node n_a, upon successfully verifying the n_d's signature on the RREP message, picks a routing fee, denoted by τ_{fee} in the range $[f_{min}, f_{max}]$, where f_{min} and f_{max} are the minimum and maximum values of the routing fees. In *Auroch* we picked the f_{min} as 1 Satoshi and f_{max} as 10 Satoshi [34]. The auctioneer n_a then computes the reservation price (*ResPrice*), according to the Equation 1. The node n_a constructs an AUC message tuple and sends this tuple along with its signature on the message (created using the temporary signing key of n_a) to all of its bidder nodes n_b in $Bidder(n_a)$ from whom it has received the initial RREQ message (Lines 8–11).

Protocol 2: Route Auctioning: Bidding Setup Phase

1 n_b maintains a set $AN(n_b) = \emptyset$
2 n_b, where $n_b \in Bidder(n_a)$, does $\forall$ AUC messages
3 **if** $\text{Verify}_{VK_{n_a}}(\text{AUC}) \rightarrow 0$ **then**
4 | Do nothing
5 **else**
6 | n_b adds each auctioneer node $AN(n_b, n_a)$ to set $AN(n_b)$
7 **if** $|AN(n_b)| == 1$ **then**
8 | n_b sets the $BidCap(n_b, P(n_d, n_a))$
9 | n_b chooses a random $\tau_1 \in [f_{min}, f_{max}]$
10 | n_b computes $BidPrice(AN(n_b, n_a))$ using Price Rule 1 in Equation 2
11 | n_b constructs BID message, n_b does $\text{Sign}_{\text{SK}_{n_b}}(\text{BID}) \rightarrow \sigma_{\text{BID}}$
12 | n_b sends BID message along with σ_{BID} to auctioneer
13 **else if** $|AN(n_b)| > 1$ **then**
14 | **for** *each* $m \in AN(n_b)$ **do**
15 | n_b sets $BidCap(n_b, AN(n_d, m))$ for each auctioneer $m \in AN(n_b)$
16 | n_b computes $BidPrice(AN(n_b, m))$ using Price Rule 1 in Equation 2 or using Price Rule 2 in Equation 3
17 | n_b constructs BID message, n_b
18 | $\text{Sign}_{\text{SK}_{n_b}}(\text{BID}) \rightarrow \sigma_{\text{BID}}$
19 | n_b sends BID message along with n_b σ_{BID} to auctioneer $AN(n_b, m)$
/* Route Auctioning: End Bidding Setup Phase */

Bidding Setup Protocol 2 This protocol handles the bidding setup phase in *Auroch*, upon receiving BID messages. n_b verifies the signature of the auctioneer node n_a on the AUC tuples. If the signature verification is unsuccessful, then the AUC tuple is discarded. Upon successfully verifying the signature of n_a on the AUC message, n_b adds auctioneer n_a to set $AN(n_b)$ that it locally maintains (Lines 1-6). If the number of auctioneers for node n_b is 1, τ_1 is chosen randomly from a specified range. n_b then determines the path capacity that it is willing to send along the path $P(n_d, n_a)$. n_b computes *BidPrice* for auctioneer n_a using Equation 2. Once the *BidPrice* has been computed, bidder n_b constructs the BID message and sends this message along with its signature to each auctioneer n_a (Lines 7-12).If the number of auctioneers for the node n_b is more than 1, n_b computes *BidPrice* for each auctioneer n_a using Equation 4. Once the *BidPrice* has been computed for each auctioneer, n_b constructs the BID message, and sends this message along with its signature to each node n_a (Lines 13-19).

Auction results and Announcing Protocol 3: Upon receiving bidding messages from all bidding nodes interested in the auction. The auctioneer n_a verifies each BID message. If the verification is successful, each bidder node $BN(n_b, n_a)$ is added to $BN(n_a)$ (Lines 1-6). If there is only one bidder in $BN(n_a)$, the auctioneer n_a updates the path $P(n_d, n_a)$ by incorporating $BN(n_b, n_a)$ into the path. If there is only one bidder in $BN(n_a)$, auctioneer node n_a updates $P(n_d, n_a)$ by adding $BN(n_b, n_a)$ into the path $P(n_d, BN(n_b, n_a))$. Subsequently, Auctioneer node n_a updates $Cap(n_d, n_a)$ into $Cap(n_d, BN(n_b, n_a))$

Protocol 3: Route Auctioning: Auction Results and Announcing Phase

/* n_a is the auctioneer */
n_a maintains a set $BN(n_a) = \emptyset$
$\forall$ BID messages, n_a checks
if Verify$_{VK_{n_b}}$(BID) $\rightarrow$ *0* **then**
  Do nothing
else
  n_a adds each bidder n_b to set $BN(n_a)$
if $|BN(n_a)|== 1$ **then**
  n_a updates $P(n_d, n_a)$ to $P(n_d, BN(n_b, n_a))$ and updates $Cap(n_d, n_a)$ to $Cap(n_d, BN(n_b, n_a))$
  n_a updates $Cost(n_d, n_a)$ to $Cost(n_d, BN(n_b, n_a))$
  n_a constructs RREP = [txid, $P(n_d, n_b), Cap(n_d, n_b), Cost(n_d, n_b)$]
  Sign$_{SK_{n_a}}$(RREP)$\rightarrow \sigma_{RREP}$ and n_a sends RREP along with σ_{RREP} to n_b
else if $|BN(n_b)|> 1$ **then**
  n_b solves Equation 6
  for *each* $BN(n_b, n_a) \in BN(n_a)$ **do**
  n_a updates $P(n_d, n_a)$ to $P(n_d, BN(n_b, n_a))$
  n_a updates $Cap(n_d, n_a)$ to $Cap(n_d, BN(n_b, n_a))$
  n_a updates $Cost(n_d, n_a)$ to $Cost(n_d, BN(n_b, n_a))$
  n_a constructs RREP = [txid, $P(n_d, n_b), Cap(n_d, n_b), Cost(n_d, n_b)$]
  n_a Sign$_{SK_{n_a}}$(RREP)$\rightarrow \sigma_{RREP}$ and n_a sends RREP along with σ_{RREP} to n_b
/* Route Auctioning: End Auction Results and Announcing Phase */

Protocol 4: Route Selection

/* Protocol 1, Protocol 2, and Protocol 3 are repeated until the source node n_s becomes a bidding node and bids for the different paths from the auctioneer nodes. */
n_s maintains the list $\mathcal{P} = \emptyset$, $\mathcal{C} = \emptyset$, and $\mathcal{F} = \emptyset$
n_s receives RREP messages
for *each* RREP *message* **do**
  IfVerify$_{VK_{n_a}}$(RREP) $\rightarrow$ 0 Do nothing
  else
    n_s adds $P(n_d, n_s)$ to $\mathcal{P}$
    n_s adds $Cap(n_d, n_s)$ to $\mathcal{C}$
    n_s adds $Cost(n_d, n_s)$ to $\mathcal{F}$
n_s solves Equation 7
/* End Route Selection */

, which is path capacity along the path $P(n_d, BN(n_b, n_a))$. Auctioneer node n_a updates $Costap(n_d, n_a)$ into $Cost(n_d, BN(n_b, n_a))$, which is the total cost of forwarding payment along the path $P(n_d, BN(n_b, n_a))$. Auctioneer node n_a updates $Cost(n_d, n_a)$ into $Cost(n_d, BN(n_b, n_a))$, which is the total cost of forwarding payment along the path $P(n_d, BN(n_b, n_a))$.Auctioneer node n_a constructs the RREP message and sends it with its signature to bidder node $BN(n_b, n_a)$ (Lines 7-12). If the auctioneer n_a has more than 1 bidder node in set $BN(n_a)$, then n_a solves Equation 6. By solving the path capacity allocation problem, Auctioneer n_a allocates the maximum capacity $MaxCap(n_d, n_a)$ in protocol 1 along the $P(n_D, n_a)$ among bidding nodes M in set $BN(n_a)$ in such way maximize his profit. for each m auctioneer in the set $BN(n_a)$,Auctioneer node n_a updates $P(n_d, n_a)$ into $P(n_d, BN(m, n_a))$ by adding bidder node $BN(m, n_a)$. Auctioneer node n_a updates $Cap(n_d, n_a)$ into $Cap(n_d, BN(m, n_a))$ Auctioneer node n_a updates $Cost(n_d, n_a)$ into $Cost(n_d, BN(m, n_a))$. Auctioneer node n_a updates $Cost(n_d, n_a)$ into $Cost(n_d, BN(m, n_a)).BN(n_b, n_a))$. Finally, Auctioneer node n_a constructs the RREP message and sends it with its signature to bidder node $P(n_d, BN(m, n_a))$ (Lines 13-20).

Route Selection Protocol 4 : This is the final stage in our construction. This protocol handles the route selection between the n_s and n_d. Upon receiving RREP messages, Node n_s verifies the signature of node n_a on each RREP message. Upon successful verification, n_s adds $P(nd, n_s)$,$Cap(n_d, n_s)$, and $Cost(n_d, n_s)$ to $\mathcal{P}$,$\mathcal{C}$, and $\mathcal{F}$, respectively. n_s allocates the partial payments among the available paths by solving the linear optimization problem Equation 7. The equation, as described earlier, minimizes the sender's overall routing cost and also maximizes the profit of each intermediate node.

Optimizations While *Auroch* provides sender privacy, the identity of the receiver is revealed during the path construction. However, since each node in *Auroch* uses a pseudonymous identity within the network, the real identity is not to any other node except the sender. These identities can be generated at periodic time intervals (which can be a system parameter) to provide an additional degree of privacy. Apart from this, other techniques such as onion routing [16] and adding dummy nodes by the receiver can be employed, which do not require any additional modifications to the structure or protocols of *Auroch*.

6 EVALUATION

6.1 Experiment Setup

We implemented *Auroch* and other routing protocols using the NetworkX library in Python [15] and used the Elliptic Curve Digital Signature Algorithm, ECDSA, [9] for signature creation and verification. ECDSA was chosen for its better efficiency. However, *Auroch* can be deployed using any digital signature algorithm. We ran our experiments on a desktop computer with Intel (*R*) Core ^{TM} i7-10700 CPU clocked at 2.90 GHz and equipped with 32 GB RAM.
Dataset: We evaluated the performance of *Auroch* on a snapshot of the Lightning Network topology from November 27, 2021 [6, 24]. The snapshot contains 18,331 nodes, and 80,918 channels with mean channel capacity 4056152.35 satoshi and median channel capacity is 1000000 satoshi (this provides us information on how funds are distributed among Lightning Network channels). In Lightning Network, the channel balance in a payment channel at the time of channel opening is publicly available on the blockchain. However, the local channel balance of nodes, which changes as a result of nodes being involved in transactions is not publicly available.

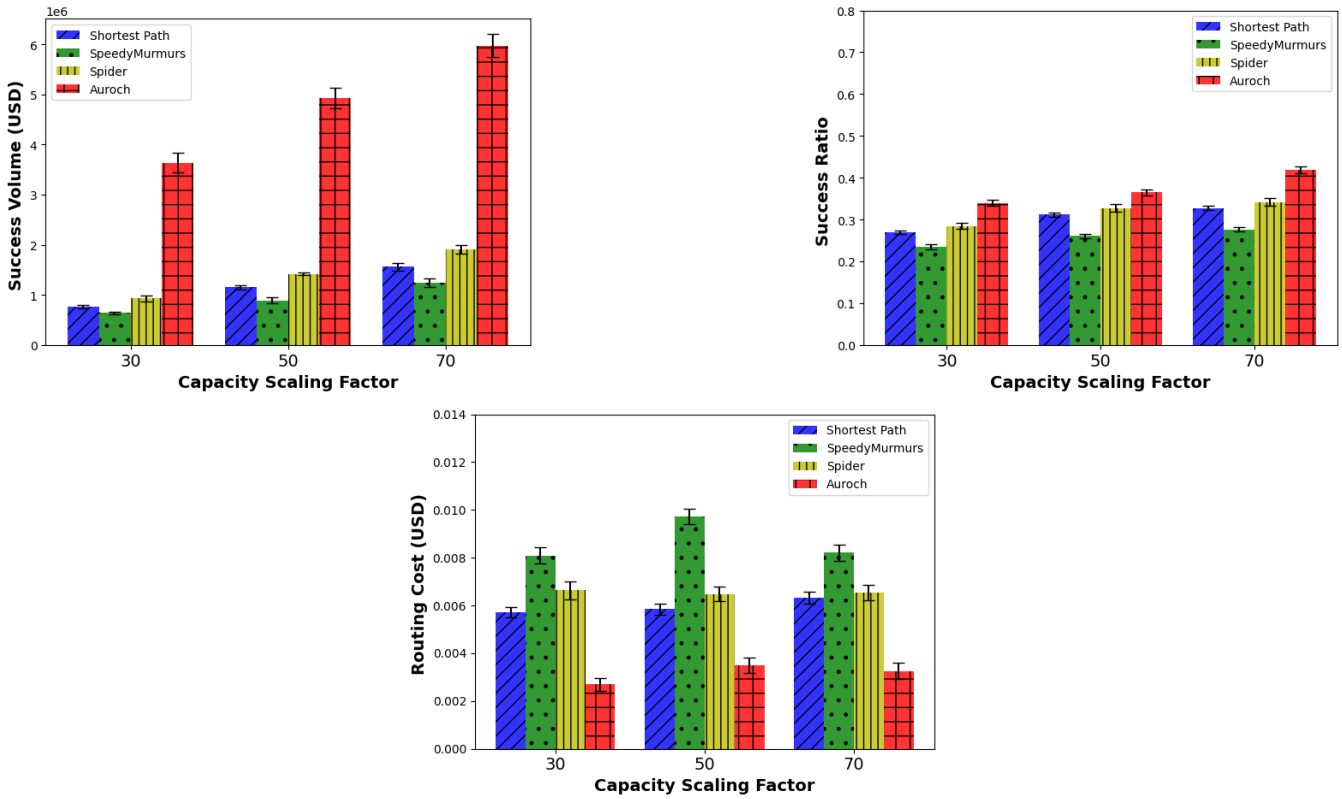


Figure 3: Performance of Routing Protocols on Lightning Network data with scaled payment channel capacities (× 30 , × 50, × 70)

Benchmarks: We implement and experimentally compare *Auroch* with several other comparable routing algorithms, specifically with shortest path routing (Dijkstra's algorithm), Speedymumurs [26] and Spider [29]. Since *Auroch* is a routing protocol that routes a single transaction along several paths, concurrency plays an important role (all the splits of the transaction would need to reach the receiver at the same time to ensure correct processing of the transaction amount). The routing protocols Spider and Speedymurmurs provide support for concurrent payments. In addition, *Auroch* was also compared to an implementation of Dijkstra's shortest path algorithm since this is currently implemented by the real-world PCN, Lightning Network. Since standard Dijkstra's algorithm does not have any notion of minimum liquidity (which is relevant for PCNs), we implemented a modified version of Dijkstra's shortest path algorithm by enumerating all paths from sender to receiver in ascending order of path length and minimum capacities, and then picking the shortest path with the minimum satisfying liquidity for each transaction. The running time of Dijkstra's algorithm depends on the priority queue implementation used [5]. NetworkX library uses a binary heap to implement the priority queue, which is appropriate for us, since for our network graph, $G(V, E)$, $|E| \ll |V|^2/log|V|$. In shortest path routing, the sender knows the entire network topology and uses source routing to find a path from herself to the receiver. Furthermore, the sender needs to know the identities of all intermediate nodes along the path and also the base fees and the rate fees charged. This is a significant drawback of using shortest path algorithms for routing in PCNs, and would apply regardless of the specific algorithm used, e.g., Dijkstra's algorithm, Bellman-Ford algorithm, etc. For all our experiments, we randomly sampled our transaction dataset to pick amounts to be routed along with the fees. We note that the SpeedyMurmurs [26] routing protocol does not have any notion of routing fees; we have augmented it with routing fees for a fair comparison with *Auroch*. For any reason, if the path chosen by the sender does not have enough liquidity or if the fees chosen by the sender is not sufficient, the transaction fails. This exemplifies the major drawback of SpeedyMurmurs in that transaction amounts (and potentially fees) for a given transaction path are randomly picked, without the sender knowing if there is sufficient liquidity along that path to the receiver. In Spider [29], the sender knows the entire network topology (similar to the shortest path algorithm) and also minimum value that can be transacted along a given path. The sender splits the total amount according to this minimum value and the sender also knows the total base fees and the rate fees charged along the path to the receiver.

Metrics: In our experiments we measure the success ratio, success volume, and the cost of routing. The success volume and success ratio are important indicators of the performance of a routing protocol. We measure the cost of routing payments to see which routing protocol is most economical to users. The success ratio is defined

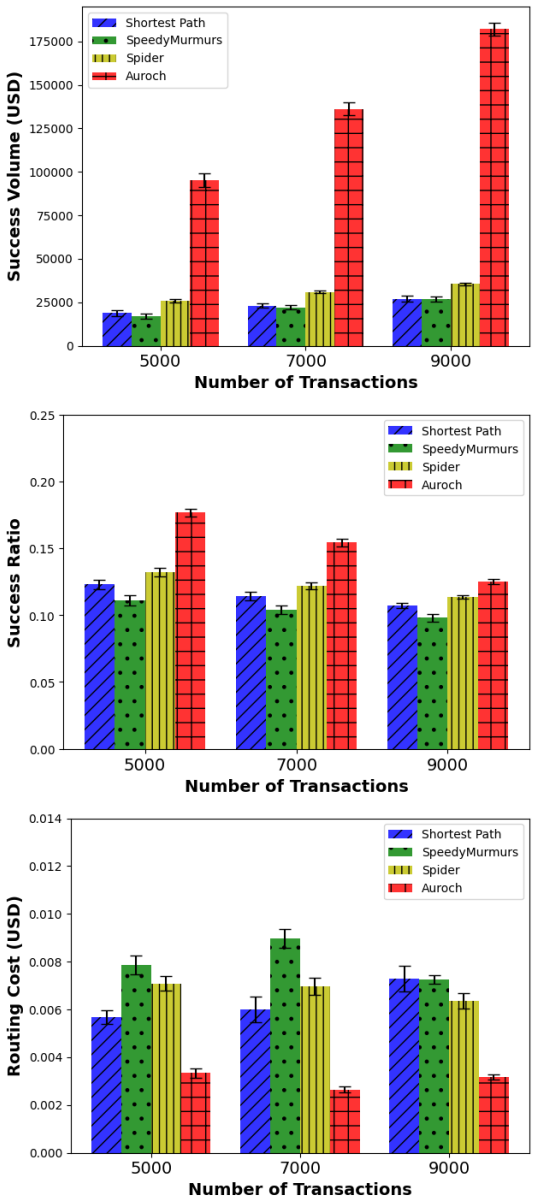


Figure 4: Performance of Routing Protocols on Lightning Network Data

as the ratio between the number of payments that were successfully routed to the total number of payments attempted. Success volume is defined as the ratio between the total volume (amount) of payments successful to the total volume (amount) of payments attempted. The cost of routing is defined as the ratio between the total cost of routing all the successful transactions at once and the volume of all the payments that were routed successfully. For an ideal routing protocol, one would expect the total transaction volume to be significantly higher than the total transaction fees, hence the ratio of the total transaction fees to the total transaction volume needs to be as low as possible.

Parameters: We set the number of transactions to 5000, 7000, and 9000, which is the approximate number of transactions recorded per month (January, February, and April, respectively) on the Ripple ledger during the year 2021. The transaction data in Ripple, such as the transaction amounts, currency, time stamp of transaction, etc are publicly available and can be accessed using the Ripple APIs [23]

To test scalability, we use number of transactions from Ripple PCN, since Lightning Network datasets have no information about the number of transaction and transaction values. In our experiments, the source and the destination nodes of a transaction are selected randomly and the transaction amount is randomly sampled from the Bitcoin trace for transaction volumes [33]. The cost of a payment from a sender to the receiver is computed for every transaction and all the paths along which all the splits of this transaction are routed. The transaction fees on each path is computed as $\sum_{i=1}^{n}$ Base Fee$_i$ + (rate Fee$_i$ $\times$ amt), where n is the total number of intermediate nodes between the sender and receiver. In our experiments, we assume the base fees along every path to be 1 satoshi and the rate fees are randomly sampled in the range [0..10] satoshis. For the various routing schemes, we set the number of disjoint paths for the Spider as 4 similar to [29], and set 3 landmarks for SpeedyMurmurs similar to [26]. We have chosen these routing protocols for our comparison since they either offer comparable security/privacy properties to *Auroch* or support multiple path payments which is one of the key aspects in the design of *Auroch*. We varied the capacity on each channel by multiplying the channel capacity by 30, 50, and 70 to study the channel capacity's impact on our metrics across all routing protocols. All experiments were averaged over 20 runs.

6.2 Results

We compare the performance of four routing protocols on our dataset by varying transaction numbers and channel capacities. Figure 4 depicts the performance of the routing protocols on 5000, 7000, and 9000 transactions. Figure 4 shows that *Auroch* has the largest success volume of all protocols, which is due to its ability to process a large volume of concurrent transactions. SpeedyMurmurs and Shortest Path protocols have the lowest success ratio among all protocols, which is due to the fact that the payment in SpeedyMurmurs happens on the fly, and the shortest path algorithm uses only a single path to transmit the payment. If there is no path with satisfying liquidity, the transaction fails, thus decreasing the success ratio. On the other hand, the *Auroch* protocol uses RREP messages, which tell the sender the path capacity along each path in a fully decentralized way, without the sender needing to know the network topology, which results in a significantly higher success ratio. Similarly, the Spider routing protocol uses probe messages to get the minimum patg capacity along the paths before sending the payment. So, using RREP messages and probing messages helps increase the success ratio.

From Figure 4, we can see that *Auroch* has the lowest routing cost among all routing protocols. Spider, SpeedyMurmurs, and Shortest path routing are up to 2.2x, 2.6x, and 2.1x, respectively, more expensive than *Auroch*. Figure 3 depicts the performance of the routing protocols when the channel capacity on each link in the networks

is varied by multiplying it with 70, 50, or 30. The purpose of multiplying the channel capacity with these values shows that *Auroch* performs better (especially in terms of success volume) when there is sufficient capacity in the channels of the nodes with their immediate neighbors. As the capacity rises, more transactions begin to succeed among all of the protocols, which is expected. According to Figure 3 *Auroch* again has the lowest routing cost of all protocols. Spider, SpeedyMurmurs, and Shortest path routing are up to 2.1x, 2.7x, and 1.8x, respectively, more expensive than *Auroch*. The lowest routing cost of *Auroch* is due to the fact that with larger channel capacities, more intermediate nodes become available for forwarding payments. The increase in the number of willing nodes and the increased channel balance result in a reduction in the routing cost. Furthermore, *Auroch* uses linear optimization to decrease the cost of routing. Overall, our experiments show that *Auroch* achieves the main goal of minimizing the transaction fees per token. When the intermediate node acts as an auctioneer, it selects the bidder node that offers the highest price. On the other hand, when the intermediate node acts as a buyer node it aims to pay the lowest price with the highest capacity. Thus *Auroch* outperforms other comparable routing protocols in terms of our chosen metrics.

6.3 Tradeoffs

In *Auroch*, the n_s uses broadcasting to find a path to the n_d. This incurs a significant amount of communication overhead. In the worst case, the maximum number of hops the RREQ message tuple will travel is equal to the diameter of the network. This overhead is better than the overhead of source routing protocol currently being implemented in Lightning Network [12], where each node stores the network topology in its local storage causing a tremendous storage overhead at each node. The snapshot of the network topology stored at each node needs to be updated every time a new node joins and leaves the PCN, which means that all the nodes in the PCN would need to be online all the time, causing wastage of resources and bandwidth or offline node would initially need to synchronize with the network before being involved in transactions, which makes instantaneous payments impossible. Sometimes, nodes in Lightning Network operate from resource constrained devices. In such cases, the storage overhead incurred will be even more. Apart from this, *Auroch* aims at reducing the overall routing cost of the n_s and also maximizes the profit of the intermediate nodes, which is not currently achieved in PCNs. In addition to these, *Auroch* introduces additional delays in the PCN due to its auctioning/bidding stages, signature creation/verification during these stages and solving the linear optimization problem to present the sender with multiple paths to the receiver. These series of operations would have to be performed for every transaction that is processed using *Auroch*. If we assume the delay (in terms of additional transaction processing time) caused by auction/bidding to be α, the delay caused by signature creation/verification during the auction/bidding stages to be β and the delay caused by solving the linear optimization problem to be γ, the total delay caused by *Auroch* (for every transaction), when deployed on a PCN such as Lightning Network would be $(\alpha + \beta + \gamma)$. This additional delay in transaction processing however, provides us with the following advantages: 1) The overall routing cost of the sender is minimized across multiple paths. 2) The profit of each intermediate node involved in the transaction is maximized. 3) *Auroch* provides the sender with a decentralized pathfindig mechanism as opposed to the centralized source routing.

7 *Auroch* SECURITY ANALYSIS

In this section, we provide a formal analysis of *Auroch* in the UC framework [2]. Protocols which are composed of multiple components are usually proven secure in UC framework [10, 11, 13, 14, 17].

We define an ideal functionality $\mathcal{F}_{\mathcal{AUROCH}}$, that consists of five functionalities: $\mathcal{F}_{init}$, $\mathcal{F}_{AUC}$, $\mathcal{F}_{Payment}$ $\mathcal{F}_{htlc}$, $\mathcal{F}_{BC}$. We use the $\mathcal{F}_{sig}$ functionality [3] and one helper functionality $\mathcal{F}_{sig}$ [3]. All the functionalities maintain three tables, *utable*, txtable, BCTable. These tables are updated by these functionalities with various tuples as required. Due to space constraints, we give the definitions of the of the ideal functionalities and the proof of the following theorem in Appendix B.

Theorem 7.1. *Let $\mathcal{F}_{\mathcal{AUROCH}}$ be an ideal functionality for Auroch. Let $\mathcal{A}$ be a probabilistic polynomial-time (PPT) adversary for Auroch, and let $\mathcal{S}$ be an ideal-world PPT simulator for $\mathcal{F}_{\mathcal{AUROCH}}$. Auroch UC-realizes $\mathcal{F}_{\mathcal{AUROCH}}$ for any PPT distinguishing environment $\mathcal{Z}$.*

8 CONCLUSION

In this work, we study the pathfinding and multipath payment routing problem in PCNs. We propose *Auroch*, a distributed pathfinding and routing protocol that provides incentives for intermediate nodes to collaborate in route formation while maximizing their profit and minimizing the total payment cost for the sender. *Auroch* also supports concurrency and takes into account the dynamic channel balance of a node. We examined *Auroch* on real-world transaction data to show its effectiveness, and we validated its security within the UC framework.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Award No. 2148358, 1914635, and the Department of Energy under Award No. DE-SC0023392. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation and the Department of Energy.

REFERENCES

[1] Benjamin Brooks and Songzi Du. 2021. Optimal auction design with common values: An informationally robust approach. *Econometrica* 89, 3 (2021), 1313–1360.

[2] Ran Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 136–145.

[3] Ran Canetti. 2004. Universally composable signature, certification, and authentication. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.* IEEE, 219–233.

[4] Yanjiao Chen, Yuyang Ran, Jingyue Zhou, Jian Zhang, and Xueluan Gong. 2021. MPCN-RP: A Routing Protocol for Blockchain-based Multi-Charge Payment Channel Networks. *IEEE Transactions on Network and Service Management* (2021).

[5] Sanjoy Dasgupta, Christos H Papadimitriou, and Umesh Virkumar Vazirani. 2008. *Algorithms*. McGraw-Hill Higher Education New York.

[6] Elias Rohrer and Julian Malliaris and Florian Tschorsch [n. d.]. Elias Rohrer and Julian Malliaris and Florian Tschorsch. https://git.tu-berlin.de/rohrer/discharged-pc-data/tree/master/snapshots

[7] Qianyun Gong, Chengjin Zhou, Le Qi, Jianbin Li, Jianzhong Zhang, and Jingdong Xu. 2021. VEIN: High scalability routing algorithm for Blockchain-based payment channel networks. In *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 43–50.

[8] Hsiang-Jen Hong, Sang-Yoon Chang, and Xiaobo Zhou. 2023. Auto-tune: An efficient autonomous multi-path payment routing algorithm for Payment Channel Networks. *Computer Networks* 225 (2023), 109659.

[9] Don Johnson, Alfred Menezes, and Scott Vanstone. 2001. The elliptic curve digital signature algorithm (ECDSA). *International journal of information security* 1 (2001), 36–63.

[10] Kartick Kolachala, Mohammed Ababneh, and Roopa Vishwanathan. 2023. RACED: Routing in Payment Channel Networks Using Distributed Hash Tables. *arXiv preprint arXiv:2311.17668* (2023).

[11] Zilin Liu, Anjia Yang, Jian Weng, Tao Li, Huang Zeng, and Xiaojian Liang. 2022. Gmhl: generalized multi-hop locks for privacy-preserving payment channel networks. *Cryptology ePrint Archive* (2022).

[12] LND. 2023. LND soruce routing. https://lightning.engineering/posts/2018-05-23-routing/.

[13] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. 2016. Silentwhispers: Enforcing security and privacy in decentralized credit networks. *Cryptology ePrint Archive* (2016).

[14] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. 2018. Anonymous multi-hop locks for blockchain scalability and interoperability. *Cryptology ePrint Archive* (2018).

[15] Networkx library [n. d.]. Networkx library. https://networkx.org/

[16] Onion routing [n. d.]. Onion routing. https://www.onion-router.net/

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

[18] Simon Parsons, Juan A Rodriguez-Aguilar, and Mark Klein. 2011. Auctions and bidding: A guide for computer scientists. *ACM Computing Surveys (CSUR)* 43, 2 (2011), 1–59.

[19] Joseph Poon and Thaddeus Dryja. 2016. The bitcoin lightning network: Scalable off-chain instant payments.

[20] Pavel Prihodko, Slava Zhigulin, Mykola Sahno, Aleksei Ostrovskiy, and Olaoluwa Osuntokun. 2016. Flare: An approach to routing in lightning network. *White Paper* (2016), 144.

[21] Raiden Network [n. d.]. Raiden Network. https://raiden.network/

[22] Ripple [n. d.]. Ripple. https://ripple.com/

[23] Ripple API [n. d.]. Ripple API. https://data.ripple.com/

[24] Elias Rohrer, Julian Malliaris, and Florian Tschorsch. 2019. Discharged payment channels: Quantifying the lightning network's resilience to topology-based attacks. In *2019 ieee european symposium on security and privacy workshops (euros&PW)*. IEEE, 347–356.

[25] Stefanie Roos, Martin Beck, and Thorsten Strufe. 2016. Anonymous addresses for efficient and resilient routing in f2f overlays. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 1–9.

[26] Stefanie Roos, Pedro Moreno-Sanchez, Aniket Kate, and Ian Goldberg. 2017. Settling payments fast and private: Efficient decentralized routing for path-based transactions. *arXiv preprint arXiv:1709.05748* (2017).

[27] István András Seres, Dániel A Nagy, Chris Buckland, and Péter Burcsi. 2019. Mixeth: efficient, trustless coin mixing service for ethereum. *Cryptology ePrint Archive* (2019).

[28] Nafiseh Sharghivand, Farnaz Derakhshan, and Nazli Siasi. 2021. A comprehensive survey on auction mechanism design for cloud/edge resource management and pricing. *IEEE Access* 9 (2021), 126502–126529.

[29] Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Kathleen Ruan, Parimarjan Negi, Lei Yang, Radhika Mittal, Giulia Fanti, and Mohammad Alizadeh. 2020. High throughput cryptocurrency routing in payment channel networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[30] Trampoline routing [n. d.]. Trampoline payments. https://lightningprivacy.com/en/blinded-trampoline

[31] P. F. Tsuchiya. 1988. The Landmark Hierarchy: A New Hierarchy for Routing in Very Large Networks. In *Symposium Proceedings on Communications Architectures and Protocols* (Stanford, California, USA) *(SIGCOMM '88)*. Association for Computing Machinery, New York, NY, USA, 35–42. https://doi.org/10.1145/52324.52329

[32] Visa [n. d.]. Visa. https://usa.visa.com/dam/VCOM/download/corporate/media/visanet-technology/visa-net-booklet.pdf. Accessed: 2023-09-18.

[33] Peng Wang, Hong Xu, Xin Jin, and Tao Wang. 2019. Flash: efficient dynamic routing for offchain networks. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*. 370–381.

[34] Philipp Zabka, Klaus-T Foerster, Stefan Schmid, and Christian Decker. 2022. Empirical evaluation of nodes and channels of the lightning network. *Pervasive and Mobile Computing* 83 (2022), 101584.

[35] Kun Zhang, Rui Wang, and Depei Qian. 2010. Aim: An auction incentive mechanism in wireless networks with opportunistic routing. In *2010 13th IEEE International Conference on Computational Science and Engineering*. IEEE, 28–33.

[36] Yang Zhang, Chonho Lee, Dusit Niyato, and Ping Wang. 2012. Auction approaches for resource allocation in wireless systems: A survey. *IEEE Communications surveys & tutorials* 15, 3 (2012), 1020–1041.

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

[38] Yuhui Zhang, Dejun Yang, and Guoliang Xue. 2019. Cheapay: An optimal algorithm for fee minimization in blockchain-based payment channel networks. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE, 1–6.

A PROTOCOLS

Protocol 5: Setup and Broadcast Protocol

1 **for** $i = 1; i \leq n; i++$ **do**
2 node i does $\mathsf{KeyGen}(1^\lambda) \rightarrow \mathsf{sk}_i, \mathsf{vk}_i$
3 node i does $\mathsf{KeyGen}(1^\lambda) \rightarrow \mathsf{SK}_i, \mathsf{VK}_i$
4 node i does $\mathsf{Sign}_{\mathsf{sk}_i}(\mathsf{VK}_i) \rightarrow \sigma_{\mathsf{VK}_i}$
5 node i calls $\mathsf{RetrieveNeighbors}(\mathsf{vk}_i) \rightarrow \mathbb{I}_i$
6 node i sends vk_i to all the nodes in $\mathbb{I}_i$
7 **for** j=1;$j \leq |\mathbb{I}_i|$;j++ **do**
8 if($\mathsf{Verify}_{\mathsf{vk}_i}(\mathsf{VK}_i, \sigma_{\mathsf{VK}_i}) \rightarrow 0$), j return $\perp$
9 n_s picks $x \leftarrow\$ \{0,1\}^\lambda$, does $H(x) \rightarrow$ txid
10 n_s sends the (txid, amt, vk_{n_s}) to the n_d via a secure out-of-band communication channel n_s constructs RREQ = (txid) and broadcasts it to all the nodes in $\mathbb{I}_{n_s}$. The nodes in $\mathbb{I}_{n_s}$ forward this tuple to their neighbors until it reaches the intended n_d

Setup and Broadcast, Protocol 5: At a high level, this protocol handles the generation of the signing and verification keypairs of a node, and the creation and transmission of a transaction identifier, txid. In this paper, we assume the existence of two pairs of keys for a node i, a long-term key pair, $\mathsf{sk_i}$ and $\mathsf{vk_i}$, a temporary key pair, $\mathsf{SK_i}$, $\mathsf{VK_i}$. The temporary verification key of a node is signed by its long-term signing key (lines 1-4). This signature is verified by all the immediate neighbors of the node i in the network (lines 7, 8). This ties the node's temporary identity to its long-term identity. Each node i uses its temporary identity to interact with non-neighboring nodes in the network. This protocol also handles the communication of the transaction details, the transaction id, txid that uniquely identifies each transaction, the amt that the n_s intends to send,

the long-term verification key of the n_s, vk_{n_s} to the n_d through a secure out-of-band communication channel (line 9-10). Finally, this protocol handles the broadcasting of the RREQ tuple, containing the txid to all the nodes in the network.

B SECURITY ANALYSIS

1) $\mathcal{F}_{init}$ functionality: This functionality is depicted in Figure 5. It handles the generation and verification of identities for all the nodes in the PCN. This functionality performs two operations **Key Generation** and **Identity Verification**. In the Key Generation phase, the $\mathcal{F}_{init}$ functionality generates the long-term and temporary signing and verification key pairs using the $\mathcal{F}_{sig}$ functionality. The purpose of creating a temporary identity is to hide the real identity of a node from its non-neighboring nodes in the network. Once the identities are generated, $\mathcal{F}_{init}$ also handles the verification of the node's temporary identity by its neighbors.

Figure 5: $\mathcal{F}_{init}$ functionality

Key Generation: Upon receiving the tuple (KeyGen, sid_i) from node i or $\mathcal{S}$, $\mathcal{F}_{init}$ forwards it to $\mathcal{F}_{sig}$. Upon receiving (Long Term Verification Key, sid_i, vk_i) and (Temporary Verification Key, sid_i, VK_i), $\mathcal{F}_{init}$ stores the tuple (sid_i, vk_i, VK_i) in the table *utable* and sends the tuple (Long Term Verification Key, sid_i, vk_i) and (Temporary Verification Key, sid_i, VK_i) to the node and the simulator $\mathcal{S}$. Upon receiving the tuple (Sign, sid_i, VK_i) from node i or $\mathcal{S}$, $\mathcal{F}_{init}$ sends the tuple (Sign, sid_i, VK_i) to $\mathcal{F}_{sig}$. If $\mathcal{F}_{sig}$ responds with (Signature, sid_i, VK_i, σ_{VK_i}), $\mathcal{F}_{init}$ updates the corresponding entry in *utable* to (sid_i, $\cdot$, $\cdot$, σ_{VK_i}) and sends the tuple (Signature, sid_i, VK_i, σ_{VK_i}) to the node and also to $\mathcal{S}$. Else $\mathcal{F}_{init}$ returns $\perp$.

Identity Verification: Upon receiving the tuple (Immediate Neighbors, $\mathbb{I}_i$) from node i or $\mathcal{S}$, $\mathcal{F}_{init}$ sends the tuple (Temporary Verification Key, VK_i) to all the nodes in $\mathbb{I}_i$. Upon receiving the tuple (Verify, sid_i, VK_i, σ_{VK_i}, vk_i) from nodes in $\mathbb{I}_i$ or $\mathcal{S}$, $\mathcal{F}_{init}$ sends the tuple (Verify, sid_i, VK_i, σ_{VK_i},vk_{sid}) to $\mathcal{F}_{sig}$. Upon receiving (Verify, sid_i, VK_i, f) from $\mathcal{F}_{sig}$, $\mathcal{F}_{init}$ updates the *utable* with (sid_i, $\cdot$, $\cdot$,$\cdot$, f) and sends the tuple (Verify, sid_i, VK_i, f) to the nodes in $\mathbb{I}_i$ and this tuple is also sent to the $\mathcal{S}$.

2) $\mathcal{F}_{AUC}$ functionality: This functionality is depicted in the Figure 6. It performs two operations, **Auction** and **Bidding**. The auction operation handles the auctioning of the maximum capacities, maxcap available in the channels from the n_d to the n_s, and the corresponding bidding of the nodes for these capacities. The functionality initially broadcasts the RREQ tuple sent by n_s to all its immediate neighbors until it reaches the n_d. Once this is done, the functionality sends an AUC tuple constructed by each node taking part in the auction to all of the node's immediate neighbors from whom the RREQ message was received. This terminates the auctioning operation. In the **Bidding** phase, every node that intends to take part in the auction will construct appropriate bidding tuples with the appropriate prices and sends them to the auctioneer.

3) $\mathcal{F}_{Payment}$ functionality: This functionality is depicted in the Figure 8, it performs two operations: **Path Selection** and **HTLC Establishment**. The path selection begins with n_d sending the details needed for HTLC establishment and payment (the digest and the corresponding preimage) to the functionality. n_s then sends in the list of all the paths $\mathcal{K}$, along which all the splits of the amt will be routed. Once the appropriate checks are performed, these details are stored by the functionality. The final operation in this is the HTLC establishment, in which the functionality helps every pair of consecutive nodes along every path from n_s to n_d establish the HTLC and complete the payment.

4) $\mathcal{F}_{htlc}$ functionality: The steps of the $\mathcal{F}_{htlc}$ functionality are straight forward. This functionality is depicted in Figure 7.

5) $\mathcal{F}_{BC}$ functionality: This functionality handles the blockchain read and write operations. The steps are described in Figure 9.

Proof. **Initialization**: The actions of the honest users, $\mathbb{H}$, $\subset [1..n]$, n is the number of nodes in the network are simulated by the simulator $\mathcal{S}$ and the actions of dishonest users, $\mathbb{D} \subset [1..n]$, are simulated by the adversary $\mathcal{A}$. For each node $i \in \mathbb{H}$, $\mathcal{S}$ generates the input tuples (Key Gen, sid_i) and sends them to the $\mathcal{F}_{init}$ functionality. It calls the $\mathcal{F}_{sig}$ functionality and forwards the tuple (Key Gen, sid_i). For each node $i \in \mathbb{H}$, the $\mathcal{F}_{init}$ sends a tuple (Long Term Verification Key, sid_i, vk_i) and (Temporary Verification Key, sid_i, VK_i) to $\mathcal{S}$. For each node $i \in \mathbb{H}$, $\mathcal{S}$, generates the input (Sign, sid_i, VK_i) and sends it to $\mathcal{F}_{init}$ functionality. $\mathcal{F}_{init}$ and forwards the tuple (Sign, sid_i, VK_i) sent by $\mathcal{S}$. Upon receiving the tuple (Signature, sid_i, VK_i, σ_i), from $\mathcal{F}_{sig}$, this tuple is forwarded to $\mathcal{S}$ by the $\mathcal{F}_{init}$ functionality. The adversary $\mathcal{A}$ generates the tuples (Long Term Verification Key, sid_j, vk_j) and (Temporary Verification Key, sid_j, VK_j), for those nodes $j \in \mathbb{D}$ that have a direct connection with the nodes $\in \mathbb{H}$ and gives them to $\mathcal{S}$. The adversary $\mathcal{A}$ also generates the tuple (Verify, VK_j, σ_{VK_j}, vk_j) for those nodes $j \in \mathbb{D}$ that have a direct connection with the nodes $\in \mathbb{H}$ and sends it to $\mathcal{S}$. $\mathcal{S}$ forwards this tuple to the $\mathcal{F}_{init}$ functionality. Upon receiving the tuple (Verify, sid_j, VK_j, f) from the $\mathcal{F}_{init}$ functionality, $\mathcal{S}$ forwards this tuple to $\mathcal{A}$. If the value of f in the tuple is ϕ or 0, the $\mathcal{F}_{init}$ functionality sends a $\perp$ to $\mathcal{S}$ who forwards this to $\mathcal{A}$.

Auction : This handles the auctioning of the maxcap between the nodes along the path from the n_s to n_d and also the sending of the RREQ tuple from the n_s to the n_d. We shall describe the RREQ sending first. Here we will have 3 cases

Case 0: n_s and $\mathbb{I}_{n_s} \in \mathbb{H}$: $\mathcal{S}$ constructs the tuple (RREQ, txid) and sends this tuple to the $\mathcal{F}_{AUC}$ functionality. It sends a *success* message back to the $\mathcal{S}$.

Case 1: $n_s \in \mathbb{H}$ and some $\mathbb{I}_{n_s} \in \mathbb{D}$: $\mathcal{S}$ constructs the tuple (RREQ, txid) and sends it to the $\mathcal{F}_{AUC}$ functionality and also to $\mathcal{A}$. It sends a *success* message to the $\mathcal{S}$.

Case 2: $n_s \in \mathbb{D}$ and some intermediate nodes in $\mathbb{H}$: $\mathcal{A}$ constructs the tuple (RREQ, txid) and sends it to $\mathcal{S}$. $\mathcal{S}$ forwards this tuple to the $\mathcal{F}_{AUC}$ functionality. The $\mathcal{F}_{AUC}$ functionality sends this tuple back to $\mathcal{S}$, who forwards this tuple to $\mathcal{A}$.

Next, the n_d sends the RREP tuple to all the nodes in $\mathbb{P}_{n_d}$. Here we have 4 cases:

Case 0: n_d and $\mathbb{I}_{n_d} \in \mathbb{H}$: $\mathcal{S}$ constructs the tuple t = (RREP, (txid,

Figure 6: $\mathcal{F}_{AUC}$ Ideal Functionality

n_s, n_d communication: Upon receiving the tuple (pay, txid, amt, vk_{n_s}, sid_{n_d}, sid_{n_s}) from n_s, the $\mathcal{F}_{AUC}$ functionality send this tuple to the $\mathcal{F}_{init}$ functionality. Upon receiving the message (RESP, Success) from the $\mathcal{F}_{init}$ functionality, the $\mathcal{F}_{AUC}$ functionality forwards the tuple (pay, txid, amt, vk_{n_s}, sid_{n_d}) to the n_d and stores the txid, amt in the txtable = (sid_{n_s}, txid, amt, $\perp$, $\perp$, $\perp$, $\perp$, $\perp$, $\perp$). If the $\mathcal{F}_{init}$ functionality returns a tuple (RESP, $\perp$), the $\mathcal{F}_{AUC}$ functionality returns a $\perp$ and aborts.

RREQ **sending**: Upon receiving the tuple (RREQ, txid, sid_{n_s}, $\mathbb{I}_{n_s}$) from the n_s, $\mathcal{F}_{AUC}$ functionality send this tuple to the $\mathcal{F}_{init}$ functionality. Upon receiving the tuple (RESP, Success) from the $\mathcal{F}_{init}$ functionality, $\mathcal{F}_{AUC}$ functionality checks if (sid_{n_s}, txid, $\cdot$, $\perp$, $\perp$, $\perp$, $\perp$, $\perp$, $\perp$) exists in txtable. If yes, the $\mathcal{F}_{AUC}$ functionality forwards the tuple (RREQ, txid) to all the nodes in $\mathbb{I}_{n_s}$. Else, the functionality.

RREP **sending to** $\mathbb{P}_{n_d}$: Upon receiving the tuple t = (RREP, (txid, VK_{n_d}, $\mathbb{P}$, amt, 0, $\sigma_{\mathsf{RREP}_{n_d}}$), $\mathcal{P}_{n_d}$) from the n_d, the $\mathcal{F}_{AUC}$ functionality checks if the tuple (sid_{n_s}, txid, amt, $\perp$, $\perp$, $\perp$, $\perp$, $\perp$, $\perp$) exists in txtable. If yes, $\mathcal{F}_{AUC}$ functionality forwards this tuple to the $\mathcal{F}_{init}$ functionality. Upon receiving the tuple (RESP, Success) from the $\mathcal{F}_{init}$ functionality, the $\mathcal{F}_{AUC}$ functionality forwards the tuple (Verify, sid_{n_d}, t, σ_{RREP}, VK_{n_d}) to the $\mathcal{F}_{sig}$ functionality. Upon receiving the tuple (Verify, sid_{n_d}, t, f) from the $\mathcal{F}_{sig}$ functionality, the $\mathcal{F}_{AUC}$ checks the value of f. If f = 0 or $f = \phi$, the functionality returns a $\perp$ and aborts. If not, the $\mathcal{F}_{AUC}$ functionality sends the tuple t = (txid, VK_{n_d}, $\mathbb{P}$, amt, 0, σ_{RREP}) to each node $g \in \mathcal{P}_{n_d}$ and stores the $\mathbb{P}$, $\mathcal{P}_{n_d}$ in the txtable = (sid_{n_s}, $\cdot$, $\cdot$, $\mathbb{P}$, $\mathcal{P}_{n_d}$, $\perp$, $\perp$, $\perp$, $\perp$).

Figure 6: $\mathcal{F}_{AUC}$ Ideal Functionality Continued

Auction: Upon receiving the tuple (AUC, txid, vk_g, $\mathsf{maxcap}_{g,g+1}$, RP_g, σ^g_{AUC}) from each node $g \in \mathcal{P}_{n_d}$, the $\mathcal{F}_{AUC}$ functionality checks if the tuple (sid_{n_s}, txid, $\cdot$, $\cdot$, $\cdot$, $\perp$, $\perp$, $\perp$, $\perp$) exists in the txtable. $\mathcal{F}_{AUC}$ forwards this tuple to the $\mathcal{F}_{init}$ functionality. For every node g in $\mathcal{P}_{n_d}$, if $\mathcal{F}_{init}$ functionality returns a tuple (RESP, Success), $\mathcal{F}_{AUC}$ functionality sends the tuple (Verify, sid_g, AUC, σ^g_{AUC}, vk_g) to $\mathcal{F}_{sig}$ functionality. Else, the $\mathcal{F}_{AUC}$ functionality returns a $\perp$ and aborts. Upon receiving the tuple (Verify, sid_g, AUC, f), $\mathcal{F}_{AUC}$ checks the value of f. If f = 0 or if $f = \phi$, the functionality returns a $\perp$ and aborts. If not, the $\mathcal{F}_{AUC}$ functionality sends the tuple (AUC, txid, vk_g, $\mathsf{maxcap}_{g,g+1}$, RP_g, σ^g_{AUC}) to the nodes $g' \in \mathbb{I}_{n_d}$.

Bidding: Upon receiving the tuple (BID, txid, $\mathsf{maxcap}_{g,g'}$, $\mathsf{BP}_{g'}$, $\sigma^{g'}_{\mathsf{BID}}$) from each node $g' \in \mathbb{I}_{n_d}$ to whom the AUC tuple was sent, the $\mathcal{F}_{AUC}$ functionality checks if the tuple (sid_{n_s}, txid, $\cdot$, $\cdot$, $\cdot$, $\perp$, $\perp$, $\perp$, $\perp$) exists in the txtable. If not, the functionality returns a $\perp$ and aborts. If yes, the $\mathcal{F}_{AUC}$ functionality sends the tuple (Verify, $sid_{g'}$, (BID, $\sigma^{g'}_{\mathsf{BID}}$, $\mathsf{vk}_{g'}$) to the $\mathcal{F}_{sig}$ functionality. Upon receiving the tuple Verify, $sid_{g'}$, BID, f). If the value of f is either 0 or ϕ, the $\mathcal{F}_{AUC}$ functionality constructs a tuple (BC Write, vk_g, t, σ^g_{BID}), where t = BID tuple, and sends this tuple to $\mathcal{F}_{BC}$ functionality. $\mathcal{F}_{AUC}$ functionality also returns a $\perp$ and aborts. If not, the functionality forwards the tuple BID, txid, $\mathsf{maxcap}_{g,g'}$, $\mathsf{BP}_{g'}$, $\sigma^{g'}_{\mathsf{BID}}$) to g. s

Bidder authentication: Upon receiving the tuple (Valid Bidders, $\mathcal{B}$, txid, sid_g) from g, the $\mathcal{F}_{AUC}$ functionality updates the txtable to store ($\cdot$, txid, $\cdot$, $\cdot$, $\cdot$, $\mathcal{B}$, $\perp$, $\perp$, $\perp$).

Final RREP **sending**: Upon receiving the tuple (RREP, txid, $\mathbb{P}$, $\mathsf{currmax}_{g',n_d}$, cost_{g',n_d}, σ_{RREP}, vk_g) from g for every node g', $\mathcal{F}_{AUC}$ retrieves the txid and $\mathcal{B}$ from txtable($\cdot$, txid, $\cdot$, $\cdot$, $\cdot$, $\mathcal{B}$, $\perp$, $\perp$, $\perp$) and checks if each node $g' \in \mathcal{B}$. If not, $\mathcal{F}_{AUC}$ returns a $\perp$ and aborts. If not, $\mathcal{F}_{AUC}$ checks if txid $\in$ txtable ($\cdot$, txid, $\cdot$, $\cdot$, $\cdot$, $\perp$, $\perp$, $\perp$). If yes, $\mathcal{F}_{AUC}$ functionality sends the tuple (Verify, sid_g, RREP, σ_{RREP}, $\mathsf{vk}_{g'}$) to $\mathcal{F}_{sig}$ functionality. Upon receiving the tuple (Verify, sid_g, RREP, f) from $\mathcal{F}_{sig}$ functionality, $\mathcal{F}_{AUC}$ checks the value of f. If f = 0 or if $f = \phi$, the functionality returns a $\perp$ and aborts.

VK_{n_d}, $\mathbb{P}_{n_d}$, amt, 0, $\sigma^{n_d}_{\mathsf{RREP}}$)) and sends this tuple to $\mathcal{F}_{AUC}$ functionality, who performs the required checks and sends this tuple back to $\mathcal{S}$.

Case 1: $n_d \in \mathbb{H}$ and some $\mathbb{I}_{n_d} \in \mathbb{D}$: $\mathcal{S}$ constructs the tuple t = (RREP, (txid, VK_{n_d}, $\mathbb{P}_{n_d}$, amt, 0, $\sigma^{n_d}_{\mathsf{RREP}}$)) and sends this tuple to $\mathcal{F}_{AUC}$ functionality and $\mathcal{A}$. $\mathcal{F}_{AUC}$ functionality performs the required checks and sends this tuple back to $\mathcal{S}$.

Case 2: $n_d \in \mathbb{D}$ and some $\mathbb{I}_{n_d} \in \mathbb{H}$: $\mathcal{A}$ constructs the tuple t = (RREP, (txid, VK_{n_d}, $\mathbb{P}_{n_d}$, amt, 0, $\sigma^{n_d}_{\mathsf{RREP}}$)) and sends it to $\mathcal{S}$, who sends this to $\mathcal{F}_{AUC}$ functionality. If all the checks performed by $\mathcal{F}_{AUC}$ functionality pass, it returns the same back to $\mathcal{S}$ who sends this to $\mathcal{A}$. If not, the functionality returns a $\perp$ to $\mathcal{S}$. **Case 3**: n_d and $\mathbb{P}_{n_d} \in \mathbb{D}$: This case is simulated by $\mathcal{A}$.

Now auction commences between every pair of nodes g and g'. Here we will have 4 cases.

Case 0: $g \in \mathbb{H}$ and $g' \in \mathbb{H}$: $\mathcal{S}$ constructs an input tuple (AUC, txid, vk_g, $\mathsf{maxcap}_{g,g+1}$, RP_g, σ^g_{AUC}) and sends it to the $\mathcal{F}_{AUC}$ functionality. Upon receiving this tuple from $\mathcal{S}$, the $\mathcal{F}_{AUC}$ functionality stores the required information and sends this tuple back to $\mathcal{S}$ who forwards this tuple to the node g'.

Case 1: $g \in \mathbb{H}$ and $g' \in \mathbb{D}$: $\mathcal{S}$ constructs an input tuple (AUC, txid, vk_g, $\mathsf{maxcap}_{g,g+1}$, RP_g, σ^g_{AUC}) and sends it to the $\mathcal{F}_{AUC}$ functionality. The $\mathcal{F}_{AUC}$ functionality stores the required information sends this tuple back to $\mathcal{S}$ who forwards this tuple to $\mathcal{A}$ that simulates the node g'.

Case 2: $g \in \mathbb{D}$ and $g' \in \mathbb{H}$: Initially $\mathcal{S}$ sends the (txid) to $\mathcal{A}$. $\mathcal{A}$ will construct the auction tuple (AUC, txid, vk_g, $\mathsf{maxcap}_{g,g+1}$, RP_g, σ^g_{AUC}) and will send it to $\mathcal{S}$, who forwards this tuple to the $\mathcal{F}_{AUC}$ functionality. If the value of $f \neq 1$, the $\mathcal{F}_{AUC}$ functionality will return a $\perp$, abort and will send this message to the $\mathcal{S}$ who in turn forwards this to $\mathcal{A}$.

Case 3: Both g and $g' \in \mathbb{D}$: This case is locally simulated by $\mathcal{A}$.

Bidding: The bidding is carried out by the node g' after it receives the auction tuple during auction from node g. Here we have 4 cases

Case 0: $g' \in \mathbb{H}$ and $g \in \mathbb{H}$: $\mathcal{S}$ constructs a tuple (BID, txid, $\mathsf{maxcap}_{g,g'}$,

$\mathsf{BP}_{g'}$, $\sigma^{g'}_{BID}$) for the node g' and sends this tuple to the $\mathcal{F}_{AUC}$ functionality. The $\mathcal{F}_{AUC}$ functionality records the required information and sends this tuple back to $\mathcal{S}$.

Case 1: $g' \in \mathbb{D}$ and $g \in \mathbb{H}$: $\mathcal{S}$ sends the (txid, $\mathsf{maxcap}_{g,g'}$) to $\mathcal{A}$. $\mathcal{A}$ generates the bidding tuple (BID, txid, $\mathsf{maxcap}_{g,g'}$, $\mathsf{BP}_{g'}$, $\sigma^{g'}_{BID}$) and sends it to the $\mathcal{S}$. $\mathcal{S}$ forwards this tuple to the $\mathcal{F}_{AUC}$ functionality. If any of the entries in the tuple are incorrect, the $\mathcal{F}_{AUC}$ returns a $\perp$ and aborts and sends the $\perp$ to $\mathcal{S}$ who in turn forwards this tuple to $\mathcal{A}$.

Figure 7: $\mathcal{F}_{htlc}$ ideal functionality

- **Initialization:** Upon receiving the tuple (Payment, vk_{Bob}, vk_{Alice}, sid_{Alice}) from $\mathcal{F}_{Payment}$, $\mathcal{F}_{htlc}$ checks if vk_{Alice}, $\mathsf{vk}_{Bob} \in$ *utable* = (sid_{Alice}, vk_{Alice} ·, ·, ·) and *utable* = (sid_{Bob}, vk_{Bob}, ·, ·, ·) respectively. If yes, $\mathcal{F}_{htlc}$ sends a message (Init OK) to Bob. Else it returns a $\perp$. Upon receiving the tuple (Payment, vk_{Bob}, vk_{Alice}, sid_{Alice}) from $\mathcal{F}_{Payment}$, $\mathcal{F}_{htlc}$ checks if vk_{Alice}, $\mathsf{vk}_{Bob} \in$ *utable* = (sid_{Alice}, vk_{Alice} ·, ·, ·) and *utable* = (sid_{Bob}, vk_{Bob}, ·, ·, ·) respectively. If yes, $\mathcal{F}_{htlc}$ sends a message (Init OK) to Alice. Else it returns a $\perp$.
- **HTLC fulfillment:** Upon receiving the tuple (HTLC tuple, vk_i, vk_{i+1}, txid, amt, Y, X) for every pair of consecutive nodes along the path of the txid from Bob to Alice, from $\mathcal{F}_{Payment}$ functionality, $\mathcal{F}_{htlc}$ checks if $H(X) = Y$, if yes, $\mathcal{F}_{htlc}$ sends a message (Success) to $\mathcal{F}_{Payment}$. Else it returns a $\perp$.

Case 2: $g' \in \mathbb{H}$ and $g \in \mathbb{D}$: $\mathcal{S}$ constructs a tuple (BID, txid, $\mathsf{maxcap}_{g,g'}$, $\mathsf{BP}_{g'}$, $\sigma^{g'}_{BID}$) for the node g' and sends this tuple to the $\mathcal{F}_{AUC}$ functionality. $\mathcal{F}_{AUC}$ functionality records the required information and sends this tuple back to $\mathcal{S}$, who in turn forwards this tuple to $\mathcal{A}$.

Valid bidders: This handles the sending of the valid bidder's tuple by the node g to the $\mathcal{F}_{AUC}$ functionality. Here we have 2 cases.

Case 0: $g \in \mathbb{H}$: $\mathcal{S}$ constructs the tuple (Valid Bidders, $\mathcal{B}$, txid, sid_g) and sends it to the $\mathcal{F}_{AUC}$ functionality. $\mathcal{F}_{AUC}$ functionality stores the required information in the txtable and returns a *success* message to $\mathcal{S}$.

Case 1: $g \in \mathbb{D}$: $\mathcal{S}$ will send the tuple (txid) to $\mathcal{A}$. $\mathcal{A}$ creates the tuple (Valid Bidders, $\mathcal{B}$, txid, sid_g) and sends it to $\mathcal{S}$, who sends it to $\mathcal{F}_{AUC}$. If any of the entries in the tuple are incorrect, $\mathcal{F}_{AUC}$ returns a $\perp$.

RREP **send**: Once the BID tuples have been received by the node g', every node g sends an RREP tuple to the node g'. Here we have 2 cases

Case 0: $g \in \mathbb{H}$: $\mathcal{S}$ constructs the tuple (RREP, txid, $\mathbb{P}$, $\mathsf{currmax}_{g',n_d}$, cost_{g',n_d}, σ_{RREP}, vk_g) for each node g' and sends this tuple to the $\mathcal{F}_{AUC}$ functionality. The $\mathcal{F}_{AUC}$ functionality sends a *success* message to $\mathcal{S}$.

Case 1: $g \in \mathbb{D}$: $\mathcal{A}$ constructs the tuple (RREP, txid, $\mathbb{P}$, $\mathsf{currmax}_{g',n_d}$, cost_{g',n_d}, σ_{RREP}, vk_g) for every node g' and sends this tuple to $\mathcal{S}$ who forwards this tuple to the $\mathcal{F}_{AUC}$ functionality. If any of the entries in the tuple are incorrect, the $\mathcal{F}_{AUC}$ returns a $\perp$ and sends this message to $\mathcal{S}$ who forwards this to $\mathcal{A}$.

Figure 8: $\mathcal{F}_{Payment}$ functionality

Path Selection: Upon receiving the tuple (HTLC Details, Y, X, txid, sid_{n_d}) from the n_d, the $\mathcal{F}_{Payment}$ functionality checks if the tuple (sid_{n_d}, txid, ·, ·, ·, ·, ·, ·, ·, ·, ·) exists in the txtable. $\mathcal{F}_{Payment}$ also checks if the tuple (sid_{n_d}, ·, ·, ·, ·) exists in the *utable*. If these checks do not pass, the $\mathcal{F}_{Payment}$ returns a $\perp$ and aborts. If all these checks pass, the $\mathcal{F}_{Payment}$ updates the txtable to store (sid_{n_s}, ·, ·, ·, ·, ·, ·, ·, ·, ·, ·, Y, X) and sends the tuple (HTLC Details, Y, X, txid, sid_{n_d}) to sid_{n_s}. Upon receiving the tuple (All Paths, $\mathcal{K}$, sid_{n_s}) from n_s, the $\mathcal{F}_{Payment}$ functionality checks if the tuple (sid_{n_s}, txid, ·, ·, ·, ·, ·, ·, ·, ·, ·, Y, X) exists in the txtable. If this check does not pass, the functionality returns a $\perp$ and aborts. If not, the functionality updates the txtable to store (sid_{n_s}, txid, ·, ·, ·, ·, ·, ·, ·, ·, ·, ·, $\mathcal{K}$).

HTLC Establishment: Upon receiving the tuple (HTLC Tuple, vk_{i}, $\mathsf{vk}_{\mathrm{i+1}}$, txid, amt, Y, X) from i, for each pair of consecutive nodes i, i + 1 along the path from the n_s to the n_d, the $\mathcal{F}_{Payment}$ functionality checks if the tuple (sid_{n_s}, txid, amt, ·, ·, ·, ·, ·, ·, ·, Y, X, ·). $\mathcal{F}_{Payment}$ then retrieves the list $\mathcal{K}$ from the tuple (sid_{n_s}, ·, ·, ·, ·, ·, ·, ·, ·, ·, ·, ·, $\mathcal{K}$) exists in the txtable and retrieves all the $\mathbb{P}$ from the $\mathcal{K}$ and checks if the nodes i and i + 1 are consecutive nodes in any of these $\mathbb{P}$ lists. If any of these checks fail, the functionality returns a $\perp$ and aborts. If all the checks pass, the tuple (HTLC Payment, vk_{i}, $\mathsf{vk}_{\mathrm{i+1}}$, txid, amt, Y, X) is sent to the $\mathcal{F}_{htlc}$ functionality. Upon receiving the messages, either (Success) or $\perp$, they will be forwarded to the node i + 1.

$g \in \mathbb{D}$: This case is simulated by $\mathcal{A}$.

Payment: This handles the actual payment between the n_s and n_d in *Auroch*. We have the following cases to consider.

Case 0: n_s, n_d and all intermediate nodes are $\in \mathbb{H}$: $\mathcal{S}$ constructs the tuple (HTLC Details, Y, X, txid, sid_{n_d}) and sends this tuple to the $\mathcal{F}_{Payment}$ functionality. The functionality returns this tuple back to $\mathcal{S}$. $\mathcal{S}$ constructs the tuple (All Paths, $\mathcal{K}$, sid_{n_s}) and send to the $\mathcal{F}_{Payment}$ functionality. For every pair of consecutive nodes i and i + 1 along each path from n_s to n_d, $\mathcal{S}$ constructs the tuple (HTLC Tuple, vk_{i}, $\mathsf{vk}_{\mathrm{i+1}}$, txid, amt, Y, X) and sends it to the $\mathcal{F}_{Payment}$ functionality. The $\mathcal{F}_{Payment}$ functionality sends this tuple to $\mathcal{F}_{htlc}$ functionality. The $\mathcal{F}_{htlc}$ functionality sends a *success* message to $\mathcal{F}_{Payment}$ functionality, which in turn forwards this message to $\mathcal{S}$. z**Case 1:** n_s, $n_d \in \mathbb{H}$ and some intermediate nodes $\in \mathbb{D}$: $\mathcal{S}$ constructs the tuple (HTLC Details, Y, X, txid, sid_{n_d}) and sends this tuple to the $\mathcal{F}_{Payment}$ functionality. The functionality returns this tuple back to $\mathcal{S}$. $\mathcal{S}$ constructs the tuple (All Paths, $\mathcal{K}$, sid_{n_s}) and send to the $\mathcal{F}_{Payment}$ functionality. For every pair of consecutive nodes, i and i + 1 along each path fzrom the n_s to the n_d, such that node i is *honest* and node i + 1 is *dishonest*, $\mathcal{S}$ (simulating the honest node i) sends the tuple (vk_{i}, Y, X) to $\mathcal{A}$. $\mathcal{A}$ simulates the HTLC payment locally and constructs the tuple (HTLC Tuple, vk_{i}, $\mathsf{vk}_{\mathrm{i+1}}$, txid, amt, Y, X) to $\mathcal{S}$. $\mathcal{S}$ forwards this tuple to $\mathcal{F}_{Payment}$ functionality. The $\mathcal{F}_{Payment}$ functionality sends this tuple to $\mathcal{F}_{htlc}$ functionality. The $\mathcal{F}_{htlc}$ functionality returns either a *success* or a $\perp$ to the $\mathcal{F}_{Payment}$, who sends this to $\mathcal{S}$. $\mathcal{S}$ sends this message to $\mathcal{A}$. Similarly, for every pair of consecutive nodes along the path from n_s to n_d, where i $\in \mathbb{D}$ and i + 1 $\in \mathbb{H}$, $\mathcal{A}$ sens the (vk_{i}) to $\mathcal{S}$. $\mathcal{S}$

constructs the tuple (HTLC Tuple, $\mathsf{vk_i}$, $\mathsf{vk_{i+1}}$, txid, amt, Y, X) and sends this to $\mathcal{F}_{Payment}$ functionality, who sends this tuple to $\mathcal{F}_{htlc}$ functionality. The $\mathcal{F}_{htlc}$ functionality either returns a *success* or $\perp$ and this sent to $\mathcal{F}_{Payment}$ functionality, who sends this to $\mathcal{S}$, who in turn sends this to $\mathcal{A}$.

Figure 9: $\mathcal{F}_{BC}$ functionality

Blockchain read: Upon receiving the tuple (BC Read, $\mathsf{vk_i}$) from a node i in the network, the $\mathcal{F}_{BC}$ functionality retrieves the successfully mined blocks from the BCTable = ($B_1, \ldots, B_j$) and sends this data to the node $\mathsf{vk_i}$.

Blockchain write: Upon receiving the tuple (BC Write, $\sigma^t_{\mathsf{vk_i}}$, (H_{txid}, ·, t, ·, ·, ·, ·, ·, ·, 1, 1, ·, ·, ·, ·, ·, ·, ·)) from a node i in the network, the $\mathcal{F}_{BC}$ functionality constructs the tuple (Verify, sid_{i}, t, $\sigma^t_{\mathsf{vk_i}}$, $\mathsf{vk_i}$) to the $\mathcal{F}_{sig}$ functionality. Upon the receiving the tuple (Verify, sid_{i}, t, f) from $\mathcal{F}_{sig}$ functionality, If the value of f is ϕ or 0, the $\mathcal{F}_{BC}$ functionality returns a $\perp$ and aborts. If not, $\mathcal{F}_{BC}$ retrieves the fees from this transaction (H_{i}, ·, t, fee, ·, ·, ·, ·, ·, ·, 1, 1, ·, ·, ·, ·, ·, ·, ·) and checks if this $fee >$ the base fees, fee'. If yes, the $\mathcal{F}_{BC}$ functionality adds this transaction to the list of pending transactions T and adds this T to the block being currently mined B_{curr} = (·, ·, ·, T). $\mathcal{F}_{BC}$ internally runs a Proof-of-Work challenge and mines a block B_{new} and this block is propagated to all the nodes in the network.

Case 2: $n_s \in \mathbb{H}$ and some intermediate nodes are $\in \mathbb{H}$ and the $n_d \in \mathbb{D}$: $\mathcal{A}$ constructs the tuple (HTLC Details, Y, X, txid, sid_{n_d}) and sends this tuple to $\mathcal{S}$, who forwards this to $\mathcal{F}_{Payment}$. $\mathcal{S}$ constructs the tuple (All Paths, $\mathcal{K}$, sid_{n_s}) and sends this tuple to $\mathcal{F}_{Payment}$ functionality. For every pair of consecutive nodes, i and i + 1 along each path from the n_s to the n_d, such that node i is *honest* and node i + 1 is *dishonest*, $\mathcal{S}$ sends the tuple ($\mathsf{vk_i}$, Y, X) to $\mathcal{A}$. $\mathcal{A}$ simulates the HTLC payment locally and constructs the tuple (HTLC Tuple, $\mathsf{vk_i}$, $\mathsf{vk_{i+1}}$, txid, amt, Y, X) to $\mathcal{S}$, who sends it to $\mathcal{F}_{Payment}$, who then forwards this to $\mathcal{F}_{htlc}$. The $\mathcal{F}_{htlc}$ functionality returns either a *success* or a $\perp$ to the $\mathcal{F}_{Payment}$, who sends this to $\mathcal{S}$. Similarly, for every pair of consecutive nodes along the path from n_s to n_d, where i $\in \mathbb{D}$ and i + 1 $\in \mathbb{H}$, $\mathcal{A}$ sens the ($\mathsf{vk_i}$) to $\mathcal{S}$. $\mathcal{S}$ constructs the tuple (HTLC Tuple, $\mathsf{vk_i}$, $\mathsf{vk_{i+1}}$, txid, amt, Y, X) and sends this to $\mathcal{F}_{Payment}$ functionality, who sends this tuple to $\mathcal{F}_{htlc}$ functionality. The $\mathcal{F}_{htlc}$ functionality either returns a *success* or $\perp$.

Case 3: $n_s \in \mathbb{H}$, n_d and some intermediate nodes are $\in \mathbb{D}$: n_s constructs the tuple (All Paths, $\mathcal{K}$, sid_{n_s}) and sends it to $\mathcal{A}$ and the $\mathcal{F}_{Payment}$ will send the tuple (HTLC Details, Y, X, txid, sid_{n_d}) and will send this tuple to $\mathcal{S}$, who sends this tuple to $\mathcal{F}_{Payment}$ functionality. For every pair of consecutive nodes, i and i + 1 from n_s to n_d, such that i $\in \mathbb{H}$ and i + 1 $\in \mathbb{D}$, $\mathcal{S}$ (simulating node i), sends the tuple (vk_i, Y, X), to $\mathcal{A}$. $\mathcal{A}$ then constructs the tuple (HTLC Tuple, $\mathsf{vk_i}$, $\mathsf{vk_{i+1}}$, txid, amt, Y, X) and sends this tuple to $\mathcal{S}$, who forwards this to $\mathcal{F}_{Payment}$, who sends it to $\mathcal{F}_{htlc}$. The $\mathcal{F}_{htlc}$ functionality can either return a *success* or $\perp$.

Case 4: $n_s \in \mathbb{H}$, n_d and some intermediate nodes are $\in \mathbb{D}$: $\mathcal{A}$ (simulating the n_d), constructs the tuple (HTLC Details, Y, X, txid, sid_{n_d}) and sends this tuple to $\mathcal{S}$. $\mathcal{S}$ sends this tuple to $\mathcal{F}_{Payment}$ functionality. $\mathcal{S}$ (simulating the honest n_s) constructs the tuple (All Paths, $\mathcal{K}$, sid_{n_s}) and sends this tuple to the $\mathcal{F}_{Payment}$ functionality. For every pair of consecutive nodes, i and i + 1 along each path from n_s to n_d, such that i $\in \mathbb{H}$ and i + 1 $\in \mathbb{D}$, $\mathcal{S}$ (simulating node i), sends the tuple (vk_i, Y, X), to $\mathcal{A}$. $\mathcal{A}$ then constructs the tuple (HTLC Tuple, $\mathsf{vk_i}$, $\mathsf{vk_{i+1}}$, txid, amt, Y, X) and sens this tuple to $\mathcal{S}$. $\mathcal{S}$ will send this tuple to the $\mathcal{F}_{Payment}$ functionality, who in turn forwards this tuple to $\mathcal{F}_{htlc}$ functionality. The $\mathcal{F}_{htlc}$ functionality can either return a *success* or $\perp$. This sent to $\mathcal{F}_{Payment}$ functionality, who sends this to $\mathcal{S}$, who in turn sends this to $\mathcal{A}$.

Case 5: $n_s \in \mathbb{D}$, some intermediate nodes in $\mathbb{H}$, $n_d \in \mathbb{D}$: $\mathcal{A}$, simulating the n_d, constructs the tuple (HTLC Details, Y, X, txid, sid_{n_d}) and sends it to $\mathcal{S}$. $\mathcal{S}$ sends this tuple to the $\mathcal{F}_{AUC}$ functionality. $\mathcal{A}$ simulating the n_s, constructs the tuple (All Paths, $\mathcal{K}$, sid_{n_s}) and sens it to $\mathcal{S}$ and $\mathcal{S}$ forwards this tuple to $\mathcal{F}_{AUC}$ functionality. If the entries in this tuple are incorrect, the functionality returns a $\perp$ and aborts and the same message is sent to $\mathcal{A}$. For every pair of consecutive nodes, i and i + 1 along each path from n_s to n_d, such that i $\in \mathbb{H}$ and i+1 $\in \mathbb{D}$, $\mathcal{S}$ (simulating node i), sends the tuple (vk_i, Y, X), to $\mathcal{A}$. $\mathcal{A}$ then constructs the tuple (HTLC Tuple, $\mathsf{vk_i}$, $\mathsf{vk_{i+1}}$, txid, amt, Y, X) and sens this tuple to $\mathcal{S}$. $\mathcal{S}$ will send this tuple to the $\mathcal{F}_{Payment}$ functionality, who in turn forwards this tuple to $\mathcal{F}_{htlc}$ functionality. The $\mathcal{F}_{htlc}$ functionality can either return a *success* or $\perp$. This sent to $\mathcal{F}_{Payment}$ functionality, who sends this to $\mathcal{S}$, who in turn sends this to $\mathcal{A}$.

Case 6: n_s and some intermediate nodes $\in \mathbb{D}$ and $n_d \in \mathbb{H}$: $\mathcal{S}$ constructs the tuple (HTLC Details, Y, X, txid, sid_{n_d}) and sends it to the $\mathcal{F}_{Payment}$ functionality. $\mathcal{A}$ constructs the tuple (All Paths, $\mathcal{K}$) and sends it to the $\mathcal{S}$, who forwards to the $\mathcal{F}_{Payment}$ functionality. If the entries in the tuple are incorrect, the $\mathcal{F}_{Payment}$ functionality returns a $\perp$. Else, For every pair of consecutive nodes, i and i+1 from n_s to n_d, $\mathcal{S}$ sends the tuple (vk_i, Y, X), to $\mathcal{A}$. $\mathcal{A}$ then constructs the tuple (HTLC Tuple, $\mathsf{vk_i}$, $\mathsf{vk_{i+1}}$, txid, amt, Y, X) and sens this tuple to $\mathcal{S}$, who in turn forwards this tuple to $\mathcal{F}_{htlc}$ functionality. The $\mathcal{F}_{htlc}$ functionality can either return a *success* or $\perp$.

□