BlAnC: Blockchain-based Anonymous and Decentralized Credit Networks

Gaurav Panwar, Satyajayant Misra, Roopa Vishwanathan
New Mexico State University
Las Cruces, New Mexico
{gpanwar,misra,roopav}@nmsu.edu

ABSTRACT

Distributed credit networks, such as Ripple [18] and Stellar [21], are becoming popular as an alternative means for financial transactions. However, the current designs do not preserve user privacy or are not truly decentralized. In this paper, we explore the creation of a distributed credit network that preserves user and transaction privacy and unlinkability. We propose BlAnC, a novel, fully decentralized blockchain-based credit network where credit transfer between a sender-receiver pair happens on demand. In BlAnC, multiple concurrent transactions can occur seamlessly, and malicious network actors that do not follow the protocols and/or disrupt operations can be identified efficiently.

We perform security analysis of our proposed protocols in the universal composability framework to demonstrate its strength, and discuss how our network handles operational dynamics. We also present preliminary experiments and scalability analyses.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**;

KEYWORDS

Distributed credit network, anonymity, transaction atomicity, blockchain.

ACM Reference Format:
Gaurav Panwar, Satyajayant Misra, Roopa Vishwanathan. 2019. BlAnC: Blockchain-based Anonymous and Decentralized Credit Networks. In *Ninth ACM Conference on Data and Application Security and Privacy (CODASPY '19), March 25–27, 2019, Richardson, TX, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3292006.3300034

1 INTRODUCTION

Credit networks are distributed systems of trust between users, where a user extends financial credit, or guarantees assets to other users whom it deems credit worthy, with the extended credit proportionate to the amount of trust that exists between the users [2, 24]. Distributed credit networks (DCNs') are essentially peer-to-peer lending networks, where users extend credit, borrow money and commodities from each other directly, while minimizing the role of banks, clearing-houses, or bourses. The rising popularity of DCNs stem from their capability to enable direct exchanges between users, sidestepping the waiting times and arbitrage fees charged by traditional, regulated financial institutions, in exchange for users accepting counter-party credit risks. In a credit network, two users, Alice and Bob can trade directly with each other, if there exists a direct trust relationship between them, otherwise a path between them through network peers, built on credit relationships between intermediate users, is established to transfer credit.

A DCN provides the basic infrastructure for building distributed payment networks, where the payment between users could be remittances of diverse nature (e.g, fiat currency, cryptocurrency, assets' transfer, such as stocks and bonds). The goal of such remittance networks is to create a distributed financial ecosystem, best exemplified by the increasingly popular Ripple payment settlement network [18].

Credit networks have found use in several applications, such as designing and securing social networks [13], Sybil tolerant networks [24], and content rating systems [8]. Popular blockchain-based payment settlement networks (e.g., Ripple [18], Stellar [21]) use credit networks as underlying infrastructure to represent credit between users. Other examples being TrustDavis [3], Bazaar [17], and Ostra [12]. Furthermore, traditional banking systems have begun integrating blockchain-based payment settlement networks such as Ripple into their set of services [19]–an increasingly popular trend.

Conceptually, a credit network can be modeled as a directed graph where users represent vertices, weighted edges represent the credit amount that a user is willing to offer its adjacent neighbor, and the directionality of the edge represents the direction of credit flow. The amount of credit between a given pair of users is usually proportional to the degree of trust that exists between them. A user can route payments to another user over a path in the network that has sufficient credit. Once a payment gets routed from a sender to a receiver, all edges along the path will get decremented by the transmitted amount.

Both centralized and decentralized credit networks currently exist. In the centralized version [12, 17], a service provider, e.g., a bank, constructs and maintains a database of all link weights, facilitates transactions between users, and performs updates to users' credit links post transactions. In the decentralized version [15, 18, 21], users maintain their own credit links, find credit routes cooperatively, and perform updates locally. Evidently, since there is no central server to manage the network/users, find paths, and route payments, operation and maintenance of such distributed credit networks is more challenging, but the design offers better privacy guarantees and is intuitively more resilient against failures. In this paper we focus on this decentralized version.

CODASPY '19, March 25–27, 2019, Richardson, TX, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6099-9/19/03…$15.00
https://doi.org/10.1145/3292006.3300034

Challenges: For broad-based acceptance and use, any credit network has to handle the following three major challenges:
(a) *Concurrency*: In a credit network, several concurrent transactions could occur (e.g., Ripple's XRP processes 1500 transactions per second), with many of them potentially using the same credit links. The network design ought to support this, while ensuring the integrity and atomicity of every transaction – ensure either all credit links get decremented along the path between sender and receiver, or none at all. This guarantees that the right receiver gets the payment, and prevents double-spending of credits.
(b) *Efficient routing*: Routing of a credit payment, requires finding of a path between a sender and receiver that has sufficient credit, in an efficient way. This needs to be done in a network where the users know only their immediate neighbors, and the network topology is dynamic due to user churn.
(c) *Privacy*: We believe that, at a minimum, a well-designed DCN needs to guarantee sender and receiver privacy (does not reveal their identities), as well as privacy of the amount transacted between them. The DCN also needs to ensure un-linkability of transactions, guarantee the privacy of the intermediate users in the path, as well as hide the network topology from adversaries. We note that today's blockchain-based networks, such as Ripple make their entire transaction history and network topology public.

Contributions: In this paper, we present BlAnC, a fully decentralized blockchain-based credit network that provides:

(1) User and transaction privacy, while providing transaction integrity, and accountability. We also allow users to dynamically choose their transaction amounts, based on current available liquidity in the network.
(2) On-demand routing, that can swiftly adapt to changing network topology, with quick on-boarding/off-boarding of users, and very low maintenance overhead.
(3) Capability for *concurrent* transactions.
(4) Distributed blockchain-based approach to publicly document transactions and identify malicious actors in transactions, while preserving both user and transaction privacy.

In essence, we propose an alternative to proposed landmarks based routing and DCN maintenance techniques [10, 20, 23], by having a subset of users facilitating transactions, termed *routing helpers* (RHs). The set of helpers can change over time, and our protocols are resilient against possible collusion of the helpers. We also discuss possible optimizations of our work.

Outline: In Section 2, we review relevant work in the area of credit networks. In Section 3, we give our system model and assumptions. In Section 4, we review the adversary model, and required privacy/security properties. In Section 5, we present our protocols. In Section 6, we present our security analysis in the universal composability framework. In Section 7, we analyze the time, space, message, and communication overheads of BlAnC. In Section 8, we present our experiments and performance analysis. In Section 9, we discuss possible optimizations and extensions to BlAnC, and in Section 10, we conclude the paper, and discuss future work.

2 RELATED WORK

Since a credit network is essentially a flow network, intuitively, one can use Ford-Fulkerson method [6], or push-relabel algorithms [7], for computing available credit on a path, but their computation costs ($O(VE^2)$, $O(V^3)$, respectively, in a graph $G(V, E)$) do not scale well to large, dynamic, distributed networks (millions of nodes).

Prihodko *et al.* [5] proposed Flare, a routing algorithm for the Lightning Network, in which each node keeps track of its k neighbors, and maintains links to *beacon* nodes. Flare reveals the value of a link to users to all nodes in the neighborhood, and works only for Bitcoin transactions. Malavolta *et al.* [11] propose payment-channel networks that make a tradeoff between privacy and concurrency; additionally their network topology is publicly known to every user in the credit network. Designing a distributed credit network, that maintains user and transaction privacy, while supporting concurrency is a challenge.

There is extensive literature on privacy-preserving transactions in Bitcoin which we do not review here, since credit networks have different structure and privacy needs as compared to cryptocurrencies, which do not require credit links or IOU paths, secure path-finding, etc. In Bitcoin and other cryptocurrencies, any user can buy/sell goods and services to other users, whereas in a credit network, users cannot transact with each other, unless they can find a path between them. Although, unlike Bitcoin and fiat currency used by banks, credit networks enable users to transact in *different currencies* expeditiously (e.g., 3-5 seconds for XRP payments, vs. 3-5 days for bank wires), and with much lower transaction fees.

Credit networks are broadly divided into centralized and decentralized architectures. In the centralized version, there has been work into reducing the reliance on the central, trusted server by using trusted hardware and oblivious computations [14, 24]. In the distributed setting, mechanisms using *landmark routing* [23] to perform route computation between users and landmark(s), and stitching the paths together to route between users, have been proposed. The landmarks–analogous to real-world banks–have less control over the network than in the centralized setting. We now discuss the prior work most relevant to this paper [10, 20].

SilentWhispers [10] presented a DCN architecture using landmark routing, in which a subset of paths between a sender and receiver is calculated, via several landmarks. At regular time intervals, each landmark starts two instances of breadth-first-search (BFS) rooted at itself. One between the sender and itself, and the other between the receiver and itself. These two paths are stitched together to form a complete path between sender and receiver. While SilentWhispers provides transaction integrity, accountability, as well as sender/receiver and transaction value privacy, it does not provide mechanisms for concurrent transactions (essential for scalability). It is also vulnerable to deadlocks, and requires users to join the network only at fixed time intervals. Additionally, prior to going offline, a user needs to hand over her signing keys, and other transaction-related data to the landmarks, which will impersonate the user during absence.

Roos *et al.* [20] presented a DCN which uses graph embedding for efficient routing, with support for concurrent transactions. The embedding algorithm constructs a rooted spanning tree of the network graph. Nodes are addressed based on their distance from the root and routing is performed based on prefixes. In [20], (a) senders pick random credit amounts to transmit along a path, without knowing whether there is adequate liquidity on the chosen path, which leads to a high rate of transaction failure, (b) there is a waiting time imposed on a user to join the network, and (c) a path is greedily chosen based on a heuristic estimate of closeness to the receiver. In a network without high dynamicity, this could lead to linkability of transactions, and eventually compromise sender/receiver privacy.

In contrast, in our approach BlAnC, the maximum credit available on a path is computed during the ***Find Route*** phase (first phase), and users can dynamically choose their transaction amount. Further, the on-boarding process does not require a user to wait. We do not pre-compute routes; all routing is done on-demand at the start of a transaction.

While [10, 20] represent progress in this area, their solutions do not provide resilience against transaction failures, capabilities such as rollbacks and timeouts, and cannot easily be adapted to real-world credit networks. We have proactively chosen to design a blockchain-based solution, to create a secure, anonymous, and distributed events ledger for BlAnC, with built-in anonymity. Thus our system can be augmented to fit in with real-world, well-regarded blockchain-based credit networks [18, 21].

3 SYSTEM MODEL

A credit network is a directed graph where the vertices of the graph represent the users or member nodes of the credit network, the weighted edges represent the flow of credit between the nodes. A directed edge with weight γ from node j to node k signifies that k has extended γ credits to j. The in-degree of k signifies the number of nodes that k has extended credit to, while the out-degree of k signifies the number of nodes that have extended credit to k. A node can lose no more than the total credit it has extended to its neighbors. Our DCN consists of nodes with credit relationships, credit senders and receivers, and a group of volunteering nodes called *routing helpers* who facilitate transactions between them. We assume all credit amounts are non-negative integers. We also assume that credit transfer between a sender-receiver pair happens over multiple paths to increase value privacy. We give our table of notations in Table 1.

Routing Helpers: We assume the existence of a dynamic set of *routing helpers* (RHs) who help route transactions (RHs *do not* know the identities of the sender, receiver, and any intermediate nodes on the path). Any well-connected node can volunteer to be an RH by writing a “volunteer” message to the Blockchain containing its public key. A sender-receiver pair creates a credit transfer path between itself using an on-demand routing protocol with the help of intermediate RHs. Credit may be distributed across multiple paths to improve unlinkability and transaction privacy. In BlAnC, the RHs help set up checkpoints, which minimize the number of rollbacks, shorten the length of a path segment along which a failed transaction (or path set-up) needs to be re-tried, and provide resilience. For simplicity, we do not discuss routing fees or mining incentives in this paper. Incorporating these into an implementation of BlAnC would be trivial, using techniques such as [4].

Blockchain: All nodes, in BlAnC are part of a Blockchain (BC). Unlike in Bitcoin, where transactions are written to the BC, in BlAnC, the miners write signed messages to the BC, converting it into a *distributed events ledger*. Each node is subscribed to the BC, so whenever a new block is written to the BC, all the nodes in the network will get notified. When a node needs to write a message,

Table 1: Notations

Variable	Definition
λ	Security parameter
RH	Routing helper
$\alpha = \{\alpha_1, \ldots, \alpha_n\}$	α is the total credit amount; each α_i is a share of α.
hopMax	Broadcast parameter
txid, txid′ **txid″, txid‴**	Transaction ids
M	Upper-bound on neighbors along a path
seg_{jk}	Path segment between nodes j, k
tS	Current timestamp
tD	Deadline (time) for the transaction to time-out
$\mathbf{currMax}_i$	Max. link weight of user i
$\mathbf{currMax}_{seg_{jk}}$	Max. link weight along seg_{jk}
cw_{jk}	Current link weight between nodes j and k
fw_{jk}	Future link weight between nodes j and k
uw_{jk}	Updated link weight between nodes j and k
H_{jk}	Hold contract between Node j and k
P_{jk}	Pay contract between Node j and k
$BC.read()$ $BC.write()$	Blockchain read/write functions
K_{ij}	Shared symmetric key between users i, j
K_{ijk}	Shared symmetric key between users i, j, k
SK_j, VK_j	Temporary signing/verification key-pair of node j
PK_j, DK_j	Encryption/decryption key-pair of node j
C_i	Ciphertext produced by user i
σ	Signature

msg, to the BC, it calls the function $BC.write(msg)$, which adds the message to the message pool, and at a later point, a miner would write the message on to the BC. Message pools are analogous to transaction pools from Bitcoin and other cryptocurrency networks.

The RHs or any nodes in the network can become miners who help in writing transactions from the message pools. The system model calls for a low mining difficulty in the credit network for near instantaneous generation of new blocks on the BC. This will facilitate fast transactions and rollbacks. As the miners themselves are part of the DCN, thus high mining complexity (proof-of-work) is not essential in BlAnC. The block chain will be used for proof of transactions (and misbehavior); any adjudication and punitive enforcement of misbehavior is out of scope of this work. BlAnC is designed for decentralized anonymity, using a database for credit link weight storage might be more efficient but it leaks private information.

Joining/leaving the network: A node which needs to join any DCN needs to find at least one network node that is willing to extend credit to, and/or receive credit from it. In BlAnC, the joining and the existing node share their pseudonymous identities and their corresponding real identities (verification keys), along with the agreed link weights for the credits between them. The new node also joins the BC network to receive update messages from the BC (including updates about RHs). A node leaving the DCN permanently just needs to set its link weights to zero and inform its neighbors to do the same for its incoming links. Any node going offline temporarily cannot be part of any ongoing transactions in the network. Before going offline, the node needs to inform its neighbors not to send any ***Find Route*** packets to it until it comes back online. We discuss handling disconnections, etc., in Section 5.5.

4 ADVERSARY MODEL AND SECURITY PROPERTIES

In our system, the adversary can adaptively corrupt a subset of users. Once user i is corrupted, its credit links will be controlled by the adversary, the adversary can misreport i's link credit value, not respond to requests, relay fraudulent messages to neighbors, and try to re-route payments to other adversary(s). An adversary can also corrupt RHs, who could possibly collude with other malicious users, but we assume a honest majority of RHs. We assume that an adversary cannot corrupt *all* users in the DCN, and thus may know partial network topology, but does not have global knowledge.

We assume that all users have a long-term verification/signing key-pair, and user i's long-term key-pair is denoted by (vk_i, sk_i). Further, all users have pseudonymous, temporary key-pairs; let us denote the temporary verification/signing key-pair of user i by (VK_i, SK_i). The temporary key-pair is signed by the long-term signing key: $\sigma \leftarrow Sign_{sk_i}(VK_i)$. This effectively ties the temporary keys (identities) to the real/longterm identity. Each user i exchanges its temporary key-pair with all of its neighbors, who in turn verify i's pair using i's long-term verification key. A user's pseudonym and temporary key-pair do not change unless there is a dispute or user failure. The temporary verification key of each RH in the system is known to all users, along with the permanent public encryption key of the RH. Sender and receiver in a transaction share each other's temporary key-pairs.

Desired Security/Privacy properties: We now outline the privacy and security properties provided by our system.

Sender/receiver privacy: An adversary will not know the real or pseudonymous identities of the sender/receiver in any successful transaction, unless she is their next-hop neighbor (all neighbors know each others' identities).

Link privacy: An adversary only knows the value of credit links adjacent to her.

Value privacy: An adversary not on the sender-receiver path does not know the amount being transacted. A corrupted node on a sender-receiver path will know the fraction of the amount transacted through her (unavoidable), but will not know the sender/receiver identities. Also, an adversary cannot compute the total credit transferred between two node pairs without compromising at least one node on all the credit fraction paths (the sender-receiver pair transfer credit concurrently along multiple paths to improve unlinkability).

Accountability: An adversary cannot re-route payments or misreport her credit link value without being detected by her honest neighbors. Malicious users violating the protocol can be identified and barred from being in the credit paths.

Integrity/Rollback: If a transaction does not go through successfully (after multiple retries), every credit links on the sender-receiver path will get rolled back to its original value. If a transaction goes through successfully, the credit links on the path will get decremented by the credit amount correctly.

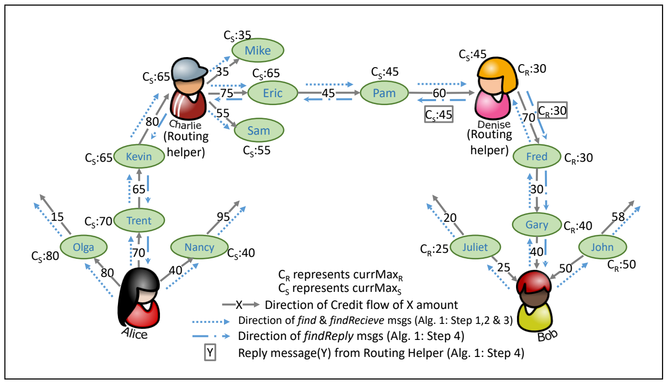

Figure 1: *Find Route* Phase: Alice and Bob use Charlie and Denise as RHs **for one of the transaction-split α_i.**

5 CONSTRUCTION OF BlAnC

The operations of BlAnC can be summed up using three broad phases: ***Find Route***, ***Hold***, and ***Pay***, which we discuss here.

5.1 *Find Route* Phase

In this phase, at a high-level a sender Alice needs to send receiver Bob an amount α, shares of which will be transmitted along different paths. Alice and Bob agree on the number of paths, n, and pick two RHs for each path (to break up the path and improve unlinkability). The maximum transmittable amount along each path, α_i (where $i \in [1..n]$), will be determined dynamically by Alice and Bob after the RHs find a path between Alice and Bob, and report to them the maximum available credit on that path. As shown in the illustration in Figure 1, Alice and Bob use RHs Charlie and Denise. The route between Alice and Bob is segmented at the two RHs: segment between Alice and Charlie (seg_{AC}), segment between Charlie and Denise (seg_{CD}), and segment between Denise and Bob (seg_{DB}).

Alice broadcasts a *find* message towards Charlie on seg_{AC}; the message is broadcasted forward by each neighbor that receives it until the copies reach Charlie. Bob performs a similar broadcast of the *findReceive* message towards Denise on seg_{DB}. Both RHs only act on the first *find* or *findReceive* messages they receive, and drop all later duplicate messages. For readability, only a single path (seg_{AC}, seg_{CD}, seg_{DB}) between Alice and Bob is shown.

In case the *find* or *findReceive* did not reach the intended routing helper, Alice or Bob respectively, can update the **hopMax** value in the tuple before re-broadcasting it. When the *find* message reaches Charlie, he retrieves the maximum credit available on seg_{AC}, $C_s = 65$, and forwards the *find* message to Denise. The max. credit available on seg_{CD} is 45, so Denise sets $C_s = 45$, and forwards a *findReply* message to Charlie, who in turn forwards the message back to Alice. The max. available credit on the path from Alice to Denise (seg_{AC}, seg_{CD}) is 45. Denise replies to Bob with maximum credit available on seg_{DB}, $C_r = 30$. Finally, Alice and Bob compute $min(C_s, C_r) = 30$.

Algorithm 1: ***Find Route*** **Phase**

```
Input   : $\alpha, n, \lambda$, hopMax, hash function $H$, public ledger BC
Output  : Maximum available credit along $n$ paths, $\alpha_1, \ldots, \alpha_n$.
Parties : Sender: Alice, Receiver: Bob
1 for $i \in [1..n]$ do
      Step 1: Alice and Bob pick RHs Charlie, Denise, broadcast
      find and findReceive tuples along $seg_{AC}$ and $seg_{DB}$
      respectively, containing $\mathbf{currMax}_s = \mathbf{currMax}_A$ and
      $\mathbf{currMax}_r = \mathbf{currMax}_B$. (see Algorithm 6)
      Step 2: Intermediate neighbors $(j, k)$ update $\mathbf{currMax}_s$ and
      $\mathbf{currMax}_r$ along the paths by setting
      $\mathbf{currMax}_{seg_{XY}} = min(cw_{jk}, \mathbf{currMax}_{seg_{XY}})$ where
      $seg_{XY} \in \{seg_{AC}, seg_{DB}\}$, and forward tuples. (see
      Algorithm 7)
      Step 3: Charlie finds a path to Denise and the max.
      available credit along $seg_{AC}$, $seg_{CD}$. (see Algorithm 8)
      Step 4: Charlie and Denise reply with the max. credit
      values, $\mathbf{currMax}_s$ and $\mathbf{currMax}_r$, to Alice and Bob
      respectively (see Algorithm 9)
      Step 5: Alice and Bob compute out-of-band,
      $\alpha_i = min(\mathbf{currMax}_s, \mathbf{currMax}_r)$.
2 end
3 If $\alpha' = sum(\alpha_1, \ldots, \alpha_n)$, such that $\alpha' < \alpha$, Alice and Bob will
  repeat Algorithm 1 until $\alpha$ is met, or will choose to transmit
  $\alpha'$.
```

Algorithm 1 presents the algorithm (see Table 1 for notations). The steps are self-explanatory. Due to space constraints, we have given the other algorithms invoked within Algorithm 1 in the Appendix A.

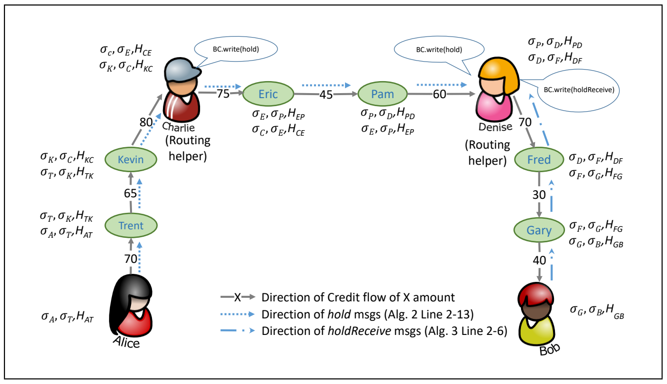
$\sigma_C, \sigma_E, H_{CE}$
$\sigma_K, \sigma_C, H_{KC}$
BC.write(hold)
Charlie
(Routing helper)
75
Eric
45
Pam
60
$\sigma_E, \sigma_P, H_{EP}$
$\sigma_C, \sigma_E, H_{CE}$
$\sigma_P, \sigma_D, H_{PD}$
$\sigma_E, \sigma_P, H_{EP}$
BC.write(hold)
$\sigma_P, \sigma_D, H_{PD}$
$\sigma_D, \sigma_F, H_{DF}$
BC.write(holdReceive)
Denise
(Routing helper)
80
Kevin
$\sigma_K, \sigma_C, H_{KC}$
$\sigma_T, \sigma_K, H_{TK}$
65
Trent
$\sigma_T, \sigma_K, H_{TK}$
$\sigma_A, \sigma_T, H_{AT}$
70
Alice
$\sigma_A, \sigma_T, H_{AT}$
70
Fred
$\sigma_D, \sigma_F, H_{DF}$
$\sigma_F, \sigma_G, H_{FG}$
30
Gary
$\sigma_F, \sigma_G, H_{FG}$
$\sigma_G, \sigma_B, H_{GB}$
40
Bob
$\sigma_G, \sigma_B, H_{GB}$
—X→ Direction of Credit flow of X amount
Direction of hold msgs (Alg. 2 Line 2-13)
Direction of holdReceive msgs (Alg. 3 Line 2-6)

Figure 2: *Hold* Phase: Between Alice and Denise on segments seg_{AC}, seg_{CD} and between Bob and Denise on segment seg_{DB}.

5.2 *Hold* Phase

After the ***Find Route*** phase, the path between Alice and Bob is identified. Now, we need to ensure that all the nodes on the path from Alice to Bob commit to the current transaction by signing contracts with their neighbors on the path. The idea is neighbors j and k (represented by (j, k)) each sign a contract which specifies their current and future link weights (after the transaction), represented as $cw_{jk}$and fw_{jk}respectively, and store each other's signatures on the contract. This contract will be written to the BC in the event of a dispute or transaction failure, thus providing accountability.

We give a pictorial representation of the ***Hold*** Phase that happens from both Alice and Bob in Figure 2. Alice constructs a *hold* tuple with $\alpha_i = 30$ and forwards it on the Alice-Denise path (in the figure, α_i is contained in every contract H_{jk}, for neighbors j, k). Each neighboring node-pair on the Alice-Denise path creates *hold* contracts between themselves. In parallel, Bob creates a *holdReceive* tuple with $\alpha_i = 30$, and forwards it on the Bob-Denise path, seg_{DB}. Each node on the Bob-Denise segment seg_{DB} segment writes the corresponding contract, signatures of its neighbor and itself on the contract (σ_j, σ_k, for neighbors j, k) into it's log file for accountability. Each RH writes a message to the BC whenever they receive *hold* and/or *holdReceive* tuples indicating the successful reception of the tuple by them.

Algorithm 2 and Algorithm 3 show the sender and receiver portions of the hold phase.

Algorithm 2: ***Hold*** **Phase: Sender and Helpers' Sub-paths**

```
Input   : Set of RHs, $\alpha = \alpha_1, \ldots, \alpha_n$, $\lambda$, hash function $H$, a
          public ledger BC, $\mathbf{txid}'$, $\mathbf{txid}''$, $K_{AD}$
Output  : hold contracts between all nodes on the path on
          $seg_{AC}$ and $seg_{CD}$
Parties : Sender: Alice, Receiver: Bob, Helpers: Charlie, Denise
1  for $\alpha_i, i \in [1..n]$ do
       /* Hold on sub-path from Alice to Charlie    */
2      begin
3          Alice picks token, $x \leftarrow \{0,1\}^\lambda$, $\mathbf{txid} \leftarrow H(x)$, shares
             token, $x$, txid with Bob; constructs tuple:
             $hold(\mathbf{txid}'||\mathbf{txid}||\alpha_i||C_A||\mathrm{tD})$, where,
             $C_A = E_{K_{AD}}(\mathrm{token}||VK_C||\mathbf{txid}||\mathrm{tS})$; sends hold to
             neighbor on path $\mathbf{txid}'$.
4          for each pair of consecutive nodes $j, k \in [1..\mathbf{M}]$ on path
             $\mathbf{txid}'$ do
5              When $k$ receives $hold(\mathbf{txid}', \mathbf{txid}, \alpha_i, C_A)$ from $j$,
                 then $k$ runs MultiSig($j \mathbin{||} SK_j \mathbin{||} VK_j \mathbin{||} k \mathbin{||} SK_k \mathbin{||}$
                 $VK_k \mathbin{||} \mathrm{tS}||cw_{jk}||fw_{jk}$); $j, k$ each locally stores
                 $(\sigma_j, \sigma_k, (H_{jk}=$ contract)) (see Algorithm 5), and
                 $k$ updates current record of $\mathbf{txid}'$ to $\mathbf{txid}$.
6          end
7          Charlie, on receiving hold, calls MultiSig(),
             and updates txid, writes a signed message to BC using
             $BC.write((VK_C||\mathbf{txid}||hold)||Sign_{SK_C}(VK_C||\mathbf{txid}||hold))$.
8      end
       /* Hold on sub-path from Charlie to Denise    */
9      begin
10         Charlie updates the tuple to
             $hold(\mathbf{txid}''||\mathbf{txid}||\alpha_i||C_A||\mathrm{tD})$, sends it to neighbor on
             $seg_{CD}$ with $\mathbf{txid}''$.
11         The intermediate nodes follow the same procedure as
             those on $seg_{AC}$, except with $\mathbf{txid}''$ instead of $\mathbf{txid}'$
             (details omitted due to space constraints).
12         Denise, on receiving hold tuple, calls MultiSig(),
             updates txid, writes a signed message to BC by calling
             $BC.write((VK_D||\mathbf{txid}||hold)||Sign_{SK_D}(VK_D||\mathbf{txid}||hold))$.
13     end
14 end
```

Alice and Bob pick a pre-image, $x \leftarrow \{0,1\}^{\lambda}$ out-of-band, and compute **txid** $\leftarrow H(x)$ (Line 3 in Algorithm 2). Note that after successful completion of the transaction, Bob will write x to the BC, which will help all nodes on the path verify that the transaction concluded successfully. Alice sends a *hold* message on the seg_{AC} to Charlie, who will in turn forward it to Denise on seg_{CD} (Line 3 in Algorithm 2). The *hold* message helps create pairwise contracts for nodes on intermediate edges on the path from Alice to Denise; the contracts are binding for them. The *hold* messages follow the path used by **txid′** on seg_{AC}, and **txid′′** on seg_{CD} in the ***Find Route*** Phase. When Charlie or Denise receive the *hold* message, they write a signed message to the BC, which indicates to all nodes on the previous segment that the *hold* message reached the target RH. In the MultiSig algorithm (Line 5, 7, 12 in Algorithm 2), nodes exchange pairwise signatures on contracts; this step is intuitive, and is given in Algorithm 5. The *hold* message will update the transaction id stored by the nodes along the path to the actual transaction id, **txid**.

Algorithm 3: *Hold* Phase: Receiver's Sub-path

```
Input   : Set of RHs, total amount $\alpha = \alpha_1, \ldots, \alpha_n$, $\lambda$, hash
          function $H$, a public ledger, BC, txid′′′, $K_{BD}$
Output  : hold contracts between all nodes on $seg_{DB}$
Parties : Sender: Alice, Receiver: Bob, Helpers: Charlie, Denise
1 for $\alpha_i, i \in [1..n]$ do
      /* Hold on sub-path from Bob to Denise */
2     begin
3         Concurrently, Bob constructs tuple:
            holdReceive(txid′′′||txid||$\alpha_i$||$C_B$||tD), where
            $C_B = E_{K_{BD}}$(token||$VK_C$||txid||tS), sends holdReceive
            tuple to next neighbor on path txid′′′.
4         The intermediate nodes follow the same procedure as
            those on $seg_{AC}$, except with txid′′′ instead of txid′.
5         Denise receives the holdReceive tuple and then creates
            hold contract with the neighbor on txid′′′ path. On
            receiving the hold from the neighbor on $seg_{CD}$
            Denise establishes a full path marked by txid from
            Alice to Bob. Finally, Denise writes a signed message
            to BC by calling $BC.write((VK_D||txid||holdReceive)||$
            $Sign_{SK_D}(VK_D||txid||holdReceive))$.
6     end
7 end
```

Similarly, Bob will send a *holdReceive* message on seg_{DB}, which will follow the path with **txid′′′** and create pairwise contracts for each link on the path (Line 3 in Algorithm 3). The nodes on seg_{DB} will also get updated with the actual transaction id, **txid**. When Denise receives the *holdReceive* message, she writes a signed message to the BC, thus indicating to all nodes on the seg_{DB} segment that the *holdReceive* message reached the target RH. The *hold* and *holdReceive* messages from Alice and Bob, respectively, contain a tD parameter. This parameter indicates the time at which the *hold* contracts will timeout if the nodes don't see a signed *hold* message (for nodes on seg_{AC} and seg_{CD}) or a signed *holdReceive* message (for nodes on seg_{DB}) corresponding to **txid** from their target RH.

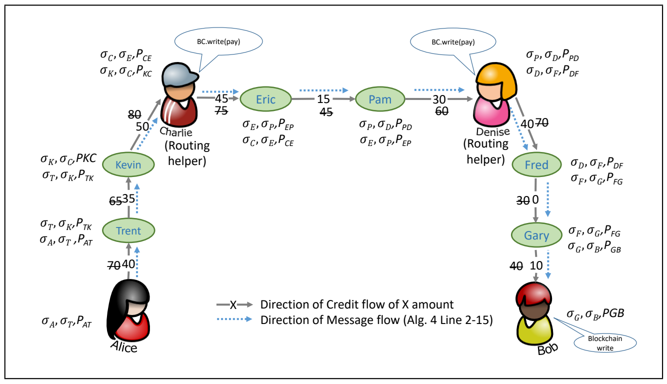
BC.write(pay)
BC.write(pay)
$\sigma_C, \sigma_E, P_{CE}$
$\sigma_K, \sigma_C, P_{KC}$
$\sigma_P, \sigma_D, P_{PD}$
$\sigma_D, \sigma_F, P_{DF}$
Eric
Pam
45
~~75~~
15
~~45~~
30
~~60~~
~~80~~
50
Charlie (Routing helper)
Denise (Routing helper)
40 ~~70~~
$\sigma_E, \sigma_P, P_{EP}$
$\sigma_C, \sigma_E, P_{CE}$
$\sigma_P, \sigma_D, P_{PD}$
$\sigma_E, \sigma_P, P_{EP}$
σ_K, σ_C, PKC
$\sigma_T, \sigma_K, P_{TK}$
Kevin
Fred
$\sigma_D, \sigma_F, P_{DF}$
$\sigma_F, \sigma_G, P_{FG}$
~~65~~ 35
~~30~~ 0
$\sigma_T, \sigma_K, P_{TK}$
$\sigma_A, \sigma_T, P_{AT}$
Trent
Gary
$\sigma_F, \sigma_G, P_{FG}$
$\sigma_G, \sigma_B, P_{GB}$
~~70~~ 40
~~40~~ 10
Direction of Credit flow of X amount
Direction of Message flow (Alg. 4 Line 2-15)
$\sigma_A, \sigma_T, P_{AT}$
Alice
σ_G, σ_B, PGB
Blockchain write
Bob

Figure 3: *Pay* Phase: Alice creates a *pay* tuple with $\alpha_i = 30$, and forwards it on the Alice-Bob path (in the figure, α_i is contained in each P_{jk} contract, for nodes j, k). Each RH writes a message to the BC whenever they receive the *pay* tuple. When Bob receives α_i credits, he writes a success message to the BC.

At the conclusion of the ***Hold*** phase, the three different paths from the ***Find Route*** phase coalesce into a single path marked by **txid** from Alice to Bob through Charlie and Denise.

5.3 *Pay* Phase

At the end of the ***Hold*** phase, all nodes on the path from Alice to Bob would have committed α_i credits to the current transaction, **txid**, by signing contracts with neighbors on the path. In the ***Pay*** phase, Alice sends a *pay* tuple along the path to Bob: *pay*(**txid**, α_i, tD) to complete the transaction. Algorithm 4 shows the steps of the ***Pay*** phase; we give a pictorial representation of the ***Pay*** phase in Figure 3.

Each node first signs *pay* contracts, corresponding to its previously signed *hold* contracts, with neighbors on the path and changes its link weights: this step is intuitive, and is given in Algorithm 5. Whenever Charlie or Denise receive the *pay* message, they write a signed message to the BC, thus indicating to all nodes on the previous segment that the *pay* message reached the target RH. The *pay* message from Alice, contains a tD parameter indicating the time at which the *pay* contracts will timeout if the nodes don't see a signed *pay* message for **txid** from their target RH.

5.4 Blockchain Operations

Because of low mining complexity in the BC, we assume that there will not be any shortage of miners. Thus ensuring timely propagation of new blocks containing latest messages from nodes within the network. The relatively higher volume of messages in BC can be dealt by creating an archived snapshot of BC at certain fixed time intervals and starting a new one-block chain. In this model, individual resource constrained user nodes need not store the entire BC, but only the compacted chain from the last snapshot. However, unconstrained devices (users, RHs) can store longer chains of the BC to act as a source of truth for older transaction data.

Whenever a new block is mined, and written onto the BC, the underlying BC protocol's consensus algorithm synchronizes it across the network. The proposed credit network can be deployed on any existing BC as long as it can accommodate certain features such as supporting individual nodes writing signed messages to the BC as opposed to writing transactions, and also have low mining complexity.

5.5 System Dynamics: Handling Timeouts, Node Failures, and Corrupt Nodes

Timeout in *Hold* Phase: All nodes know who the target RH on their respective segment is, e.g., Charlie for seg_{AC}, and so on. If there are no *hold* messages written on BC associated with the current **txid** by either RHs, this means that the *hold* messages timed out in the first segment and never reached Charlie. In this scenario, *hold* contracts are dropped by all nodes on the path and Alice has to retry the transaction. If nodes time out during the ***Hold*** phase and see a *hold* message on BC from their target RH, they do not drop their *hold* contracts, they wait for the transaction to be retried and completed in another segment.

However, if the nodes on any segment do not see a *hold* message from their target RH before timing out, they drop the *hold* contracts and reservation on their links, since the transaction timed-out in their segment, and the ***Find Route*** phase will be retried on that segment. All the nodes on the specific timed-out segment also publish the dropped contracts onto the BC. This will expose the offending node's (node which caused the time-out) ephemeral identity and thus its neighbors will not forward the find tuple to this node for the current transaction when the ***Find Route*** phase is retried in current segment. The offending node's privacy in the network is still maintained and only its immediate neighbors find out that the node timed-out in the current transaction. The offending node could be an RH, then the sender-receiver can try again or abort the transaction and start with a new set of RHs.

To illustrate, if Charlie had written the *hold* message to BC and Denise did not, then Charlie will retry for seg_{CD} by repeating the ***Find Route*** phase, and ***Hold*** phase to Denise, to find an alternate path. If the timeout occurs after Denise has written a *hold* message on the BC, but the *holdReceive* message is not complete, then Bob will retry the transaction for seg_{DB} on its end.

Timeout in Pay Phase: On timeout, each node j on a path **txid** on the timed-out segment, will call $BC.write(\mathbf{txid}||H_{jk}||P_{jk})$ if they had a *pay* contract or just $BC.write(H_{jk})$ if they did not receive a *pay* message from neighboring node. In the pay phase, seg_{AC} or seg_{CD} time-out if target RH (Charlie or Denise, respectively) did not write a signed *pay* message to the BC indicating successful reception of *pay* message. Segment seg_{DB} times-out if nodes on the segment do not see a **success** message on the BC from Bob, with a correct pre-image for **txid**. When the current transaction cannot be completed because either the timeout occurred on seg_{AC}, or if there are no alternate viable paths on seg_{CD} or seg_{DB}, Alice or Bob can abort the transaction. To abort transaction and initiate a rollback of any changes in the network (from *pay* contracts affecting link weights) tied to **txid**, Bob or Alice write the tuple: (**txid**, x, **failure** − **rollback**) to the BC using $BC.write$. All nodes on path should delete *hold* contracts and revert back to previous link weights if they had any *pay* contracts associated with transaction **txid**.

Alternatively, if the timeout occurred on seg_{CD} or seg_{DB}, then Charlie or Denise, respectively, will retry to find an alternate path. Since all contracts were written to the BC, all honest nodes in the path know which node timed out the transaction (faulty node), either by malicious behavior or by going offline, the honest nodes will route retry packets to neighbors other than the identified faulty node to prevent subsequent failures. If Charlie wrote the *pay* message to the BC, then Charlie will retry the ***Find Route***, and ***Hold*** phases to Denise to find an alternative path, before retrying the ***Pay*** phase again.

In the absence of malicious nodes in the network, only six messages will be written to the BC. Three messages are written by the RHs after the ***Hold*** and two message in the ***Pay*** phases. The

Algorithm 4: *Pay* Phase

```
Input   : Set of RHs, total amount $\alpha = \alpha_1, \dots, \alpha_n$, $\lambda$, hash
          function $H$, a public ledger, BC, txid
Output  : Updated link weights and corresponding pay
          contracts on each link from Sender to Receiver
          equivalent to transaction amount
Parties : Sender: Alice, Receiver: Bob, Helpers: Charlie, Denise
1  for $\alpha_i, i \in [1..n]$ do
       /* Pay on sub-path from Alice to Charlie */
2      begin
3          Alice constructs tuple $pay(\mathbf{txid}||\alpha_i||tD)$, sends pay
             tuple to next neighbor on path txid
4          for each pair of consecutive nodes $j, k \in [1..\mathbf{M}]$ in
             $seg_{AC}$ which have txid do
5              When $k$ receives $pay(\mathbf{txid}||\alpha_i||tD)$ from $j$, $k$ runs
                 MultiSig($j \,||\, SK_j \,||\, VK_j \,||\, k \,||\, SK_k \,||\, VK_k \,||$
                 $tS||cw_{jk}||uw_{jk}$) (see Algorithm 5). Nodes $j, k$
                 each locally store ($\sigma_j, \sigma_k$, ($P_{jk}$= contract)).
6          end
7          On receiving pay tuple, after calling Multi-
             Sig(), Charlie writes a signed message to BC by calling
             $BC.write((VK_C||\mathbf{txid}||pay)||\mathrm{Sign}_{SK_C}(VK_C||\mathbf{txid}||pay))$.
8      end
       /* Pay on sub-path from Charlie to Denise */
9      begin
10         Charlie forwards pay tuple to the next neighbor on
             txid path towards Denise.
11         Intermediate nodes follow the same steps as those on
             $seg_{AC}$. On receiving the pay tuple, and after calling
             MultiSig(), Denise writes a signed message to BC by
             calling
             $BC.write((VK_D||\mathbf{txid}||pay)||\mathrm{Sign}_{SK_D}(VK_D||\mathbf{txid}||pay))$.
             Denise forwards pay tuple to the next neighbor on
             txid path towards Bob.
12     end
       /* Pay on sub-path from Denise to Bob */
13     begin
14         Intermediate nodes on $seg_{DB}$ follow the same steps as
             those on the other segments. On receiving the pay
             tuple, after calling MultiSig(), Bob writes a success
             message to BC by calling $BC.write(\mathbf{txid}||x||\mathbf{success})$.
15     end
16 end
```

sixth message is the **success** message written by Bob to the BC informing the rest of the nodes on the path about the transaction being successful. Since no node involved in the transaction exposed their identity, there is no need to change any node's pseudonymous identities. However, if the transaction was retried, that is, a timeout occurred, all nodes involved in the transaction will need to update their pseudonymous identities and share new pseudonymous identities with each other and their neighbors. This rekeying will help reduce linkability between transactions as now all the nodes' previous pseudonymous identities are in the BC.

Algorithm 5: Multisig Exchange

```
Input  : $j, SK_j, VK_j, k, SK_k, VK_k, cw_{jk}, \gamma \in \{fw_{jk}, uw_{jk}\}$, **txid**, tS
Output : Tuple ($\sigma_j, \sigma_k$, contract) stored at node $j$ and $k$
Parties: Node $j$ and $k$
1 $j$ sends $\sigma_j \leftarrow \text{Sign}_{SK_j}(\text{contract} = (cw_{jk}, \gamma), \textbf{txid}, \text{tS})$ to $k$
2 $k$ sends $\sigma_k \leftarrow \text{Sign}_{SK_k}(\text{contract} = (cw_{jk}, \gamma), \textbf{txid}, \text{tS})$ to $j$
3 if $Verify_{VK_k}(\text{contract}||\sigma_k) \overset{?}{\leftarrow} 1$ then
4     $j$ stores $(\sigma_j||\sigma_k||\text{contract})$
5 end
6 if $Verify_{VK_j}(\text{contract}||\sigma_j) \overset{?}{\leftarrow} 1$ then
7     $k$ stores $(\sigma_j||\sigma_k||\text{contract})$
8 end
```

Malicious RHs: In case of misbehaving RHs where the RHs neglects to write *hold/pay* tuple reception messages to the BC, other nodes on the path would timeout. They would then dump all the *hold/pay* contracts for the given transactions on to the BC. This would show that all nodes on the path went through with the transaction and it was the misbehaving RH who did not update the transaction on the BC.

There is a possibility of an RH changing its public identity and coming back as a new one after it is identified as malicious. However, if users choose well-known RHs (e.g., one that has written many transactions to the BC), then the impact of such a malicious RH can be significantly mitigated. Even in the presence of misbehaving RHs the sender/receiver do not end up losing any credits as the transaction will either get re-routed or aborted in case of failure.

6 SECURITY ANALYSIS

We prove the security of our constructions in the Universal Composability (UC) framework [1] which is a well-known framework used to analyze the security of distributed protocols. The UC paradigm elegantly captures the conditions under which a given distributed protocol is secure, by comparing it to an ideal realization of the protocol. To this end, the UC framework defines two "worlds": the real-world, where the protocol, π to be proved secure runs, and the ideal-world, where the entire protocol, ϕ is executed by an ideal, trusted functionality, where all users only talk to the ideal functionality via secure and authenticated channels. The goal then is to prove that no distinguishing algorithm, commonly called as "environment", $\mathcal{Z}$, can successfully distinguish between the execution (EXEC) of the two worlds. The notion of UC security is captured by the pair of definitions below:

Table 2: Asymptotic Complexities: n denotes the number of shares of a payment (in Ripple [18], the max. number of paths, and hence n, for a single transaction is 7), d denotes node degree, k is the number of nodes on a single path ($k \subseteq M$, $|k| << |M|$), c is the max. path length between sender and receiver (from Ripple [18], the max. path length is 10).

Phases	Time	Space	Messages		
			Regular users	Charlie (RH)	Denise (RH)
Find Phase	$O(n)$	$O(n)$	$O(d^c \cdot n)$	$O(d^c \cdot n)$	$O(k \cdot n)$
Hold Phase	$O(k \cdot n)$	$O(k \cdot n)$	$O(k \cdot n)$	$O(n)$	$O(n)$
Pay Phase	$O(k \cdot n)$	$O(k \cdot n)$	$O(k \cdot n)$	$O(n)$	$O(n)$

Definition 6.1. (UC-emulation [1]) Let π and ϕ be probabilistic polynomial-time (PPT) protocols. We say that π UC-emulates ϕ if for any PPT adversary $\mathcal{A}$ there exists a PPT adversary $\mathcal{S}$ such that for any balanced PPT environment $\mathcal{Z}$ we have

$$\text{EXEC}_{\phi,\mathcal{S},\mathcal{Z}} \approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$$

Definition 6.2. (UC-realization [1]) Let $\mathcal{F}$ be an ideal functionality and let π be a protocol. We say that π UC-realizes $\mathcal{F}$ if π UC-emulates the ideal protocol for $\mathcal{F}$.

We define a distributed credit network functionality $\mathcal{F}_{\mathsf{DCN}}$ in the ideal world, which consists of $\mathcal{F}_{\mathsf{FindRoute}}$, $\mathcal{F}_{\mathsf{Hold}}$, $\mathcal{F}_{\mathsf{Pay}}$, and $\mathcal{F}_{\mathsf{BC}}$. An adversary can corrupt regular users and routing helpers at any time, upon which the user's responses to queries by $\mathcal{F}_{\mathsf{DCN}}$ will be generated by the adversary. We assume $\mathcal{F}_{\mathsf{DCN}}$ maintains an adjacency matrix of all users in the network, where the entries of the matrix are the link weights, which is regularly updated when $\mathcal{F}_{\mathsf{FindRoute}}$ and $\mathcal{F}_{\mathsf{Pay}}$ are called. The definitions of $\mathcal{F}_{\mathsf{FindRoute}}$, $\mathcal{F}_{\mathsf{Hold}}$, $\mathcal{F}_{\mathsf{Pay}}$, and $\mathcal{F}_{\mathsf{BC}}$, discussion about their design choices and correctness, and the proof of the following theorem is available in the Section 12.

Theorem 6.3. *Let $\mathcal{F}_{\mathsf{DCN}}$ be an ideal functionality for* BlAnC. *Let $\mathcal{A}$ be a probabilistic polynomial-time (PPT) adversary for* BlAnC, *and let $\mathcal{S}$ be the ideal-world PPT simulator for $\mathcal{F}_{\mathsf{DCN}}$.* BlAnC *UC-realizes $\mathcal{F}_{\mathsf{DCN}}$, for any PPT distinguishing environment $\mathcal{Z}$.*

Sketch: At a high level, the proof shows that no PPT distinguishing environment $\mathcal{Z}$ can distinguish between the outputs of the ideal-world simulator, $\mathcal{S}$, and a BlAnC adversary $\mathcal{A}$. Ideal-world $\mathcal{S}$ mirrors the actions of a real-world $\mathcal{A}$, and we show that if $\mathcal{A}$ cheats in the real-world, $\mathcal{S}$ would also break the security of the $\mathcal{F}_{\mathsf{DCN}}$, which is not possible.

7 SCALABILITY METRICS

In this section, we analyze the performance of our system with respect to time, space, message, and communication complexities. Time is measured in terms of the average execution time of a cryptographic operation, the space is measured in terms of the total storage required, the message complexity is measured in number of messages, in the worst case, and the communication complexity is the number of bytes of information transmitted. Table 2 shows the asymptotic time, space and message complexities. Table 3 shows the number of encryptions, decryptions, signatures, and hashes at each node during the ***Find Route***, ***Hold***, and ***Pay*** phases. Table 4 shows the communication complexity in bytes at different nodes.

Table 3: n is the number of shares, number of cryptographic operations at a node: E: no. of encryptions, D: no. of decryptions, S: no. of signatures, V: no. of verifications, H: no. of hashes.

Phases	Sender	Receiver	RH
Find Phase	E: $2n$, D: $2n$, H: $3n$	E: $2n$, D: $2n$, H: $3n$	E: $2n$, D: $2n$, S: n, H; n
Hold Phase	E: n, S: $2n$, V: $2n$	E: n, S: $2n$, V: $2n$	D: $2n$, S: $2n$, V: $2n$
Pay Phase	S: $2n$, V: $2n$	S: $2n$, V: $2n$	S: $2n$, V: $2n$

Table 4: Worst case communication complexity (in terms of message size): Using RSA-2048 for PKI, ECDSA signatures (72 bytes), AES-256 for symmetric key encryption and SHA-256 for token/txids' generation.

Type of Message	Size	Phase
find Tuple	166 *bytes*	***Find Route*** Phase
findReceive Tuple	134 *bytes*	***Find Route*** Phase
findReply Tuple	240 *bytes*	***Find Route*** Phase
hold Tuple	272 *bytes*	***Hold*** Phase
holdReceive Tuple	272 *bytes*	***Hold*** Phase
pay Tuple	80 *bytes*	***Pay*** Phase
BC.write hold	172 *bytes*	***Hold*** Phase
BC.write holdReceive	179 *bytes*	***Hold*** Phase
BC.write pay	171 *bytes*	***Pay*** Phase

Table 5: Emulation results for crypto operations in BlAnC

Cryptographic Operation	Find Phase	Hold Phase	Pay Phase
RSA-2048 Encrypt Time	202.78 *us*	NA	NA
RSA-2048 Decrypt Time	2.63 *ms*	NA	NA
AES-256 Encrypt Time	4.54 *us*	4.54 *us*	NA
AES-256 Decrypt Time	4.08 *us*	4.08 *us*	NA
ECDSA Sign Time	192.22 *us*	6.38 *ms*	6.17 *ms*
ECDSA Verify Time	1.10 *ms*	31.59 *ms*	31.59 *ms*
SHA-256 Hash Time	24.36 *us*	8.12 *us*	NA

Joining the network: When a new node joins the credit network, it shares pseudonymous keys, verification keys, and link weights with nodes that it will be connected to in the network and stores these values. This is a one time setup cost and is linear in the number of neighbors the new node will have in the network. The node also joins the blockchain which incurs a constant time/space overhead and is a one time setup cost.

Key exchange after timeout: When a transaction times out, all nodes involved in the transaction would have published their *hold* and *pay* contracts to the BC, exposing their pseudonyms. After termination of a timed out transaction, regardless of whether it was finally successful or aborted, all involved nodes need to establish new pseudonyms with their neighbors. The time complexity of this step is linear in the number of nodes in the path and degree of each node, $O(k \cdot d)$, where d is maximum node degree in the DCN.

8 EXPERIMENTS AND EVALUATION

The cryptographic operations used in the protocol, which are AES-256, SHA-256, RSA-2048 and ECDSA were implemented using C++ Open-SSL libraries [22]. The simulations were performed on a desktop class machine with Intel(R) Core(TM) i7-7600U CPU @ 2.80GHz and 8GB RAM. We use ns-3 [16], a discrete event network simulator to test BlAnC. The simulations were run on a 100 node network with the nodes connected over WiFi. Based on the fact that the Internet's diameter is around 18, in our simulation setup, the sender and receiver are at a distance of 15 hops away from each other, on a path passing through the two RH. This distance accounts for path stretch from Alice to Bob when using two RHs. The network's physical layer delay characteristics are set to the Constant Speed Propagation Delay Model and loss characteristics are set to the Log Distance Propagation Loss Model, both available in the ns-3 codebase. The channel coding was set to Orthogonal frequency-division multiplexing at a data rate of 6 Mbps. The simulations were run with multiple concurrent transactions taking place at the same time. A total of 200 transactions were simulated.

Table 5 shows the timings for the cryptographic operations performed by nodes on a transaction path for the different phases during a BlAnC transaction (emulated on the same desktop class machine). As reflected in the figure, the cryptographic operations in the ***Hold*** phase and ***Pay*** phases contribute very little to the time delay, that is, ~ 37 *ms* as opposed to 4 *ms* in the ***Find Route*** phase. The time delay in the ***Find Route*** phase is largely attributed to the ad hoc, on demand path finding technique of BlAnC. This delay is a trade-off to ensure that the privacy of sender and receiver is protected as each transaction triggers a new set of path searches and the paths to chosen RHs are not pre-determined.

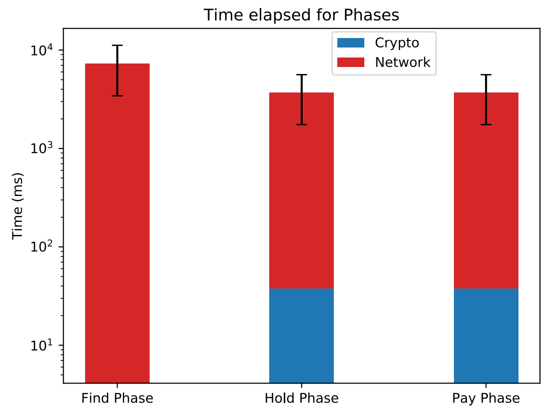

Figure 4: Illustration of average time delay for *Find Route* , *Hold* and *Pay* phases during a BlAnC **transaction due to network and cryptographic operations.**

Figure 4 shows that majority of the time delay in BlAnC comes from the network when compared to delay incurred due to cryptographic operations. The error bars represent the standard deviations of the delay. From the ns-3 simulations, we observed that the total time taken by the ***Find Route*** phase to conclude was on average 7.303 *secs*. This is the time taken by Alice and Bob to determine the maximum value of α_i credit that can flow between Alice and Bob in the chosen path for the transaction in question. The delay of 7.303 secs includes network delay, while the delay contributed by cryptographic operations was approximately 4.158 *ms*. In comparison to SilentWhispers [10] which takes 1.349 seconds (the authors only measured cryptographic costs) to determine α_i on a path of length 10 *hops* (BlAnC is three orders of magnitude faster).

The ***Hold*** and ***Pay*** phases took a total duration of 1.253 secs each. So a complete transaction on average takes 9.809 secs to finish. We did not include the time delays for the BC as those delays would not affect the flow of credit from sender to receiver unless there is an occurrence of a timeout during the transaction.

9 OPTIMIZATIONS

In the ***Find Route*** phase, the cost of finding a path from sender/receiver to their respective RHs, in the worst case could be $O(d^c)$, where d is the maximum degree of a node on the path, and c is the maximum hop count. In practice, c could be set as the maximum length of a path between two nodes, which, according to empirical data collected from Ripple's datasets is 10 [10, 18]. This is a one-time cost, which, we believe, will be amortized over time by having the sender/receiver store information about the paths to their respective helpers, over the course of their transactions. The sender/receiver would then only have to follow a fixed path to their RH, and would incur a cost of $O(d^c)$ infrequently. Another way to optimize this cost would be for every node in the network to choose a random d', such that $d' < d$, and send find probes only to the set of neighbors in d'.

Each node in the credit network that is involved in a transaction, could keep track of the identity of the RH and the interface it reached the RH from. By using this information as a forwarding table, the number of broadcasts could be decreased in a stable credit network where the links state do not vary frequently. Broadcasts could be used in-case of a stale forwarding table, in which case the sender would retry the ***Find Route*** phase with a broadcast instead of a directed *find* message. This would also reduce the cost of the protocol in the ***Find Route*** phase if the intermediate nodes do not need to use broadcasts and already have a path available to a known RH. Each node could also build a history of all the RHs it has used for prior transactions, and prefer to use the ones with which the transactions completed successfully instead of trying to send payment through new RHs.

Graph embedding, as used in [20] could be used in our ***Find Route*** phase to construct an optimal path between the sender/receiver and their respective RHs. One could construct a spanning tree of the network rooted at the RHs, and either used tree-base embedded routing (strictly following the edges of the spanning tree), or a more flexible approach, where one greedily chooses the shortest path between two nodes, regardless of whether the shortest path uses edges present in the spanning tree or not.

Since the BC is being used to publish transaction-related messages, the storage of the BC can become challenging, along with the scaling of the DCN. To tackle this problem, the BC can be compressed at regular time epochs as we discussed before. At the end of a time epoch, all nodes would cease transacting and make sure all the payments and links are settled. At this point, the BC can be wiped off and all old data can be compressed and stored with certain nodes which have enough storage, and are willing to store the older BC. The hash of the old BC can be provided at the start of the new BC linking the two together. At the start of the new epoch, all RHs need to declare themselves as RHs again, as there is no historical information available to new nodes joining the credit network in the new epoch.

10 CONCLUSIONS AND FUTURE WORK

In this paper, we propose BlAnC, a novel, fully decentralized blockchain-based credit network that preserves user and transaction anonymity, enables on-demand and concurrent transactions to happen seamlessly, and can identify malicious network actors that do not follow the protocols and disrupt operations. We performed security analysis to demonstrate BlAnC's strength and presented scalability metrics. Simulation/emulation-based analysis of the latency of transactions in the DCN demonstrate BlAnC's scalability.

In the future, we intend to implement BlAnC in a real-world testbed like Hyperledger [9] and test impact of real-world network dynamics on the protocols' stability and scalability.

11 ACKNOWLEDGEMENTS

Research supported by NSF awards #1800088; #1719342; #1345232; EPSCoR Cooperative agreement OIA-1757207; and ARO grant W911NF-07-2-0027. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the federal government.

REFERENCES

[1] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS*. 136–145.
[2] P. Dandekar, A. Goel, R. Govindan, and I. Post. 2011. Liquidity in credit networks: a little trust goes a long way. In *Proceedings of ACM Conference on Electronic Commerce (EC)*. 147–156.
[3] D. B. DeFigueiredo and E. T. Barr. 2005. TrustDavis: A Non-Exploitable Online Reputation System. In *IEEE International Conference on E-Commerce Technology (CEC 2005)*. 274–283.
[4] Felix Engelmann, Henning Kopp, Frank Kargl, Florian Glaser, and Christof Weinhardt. 2017. Towards an Economic Analysis of Routing in Payment Channel Networks. In *Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers (SERIAL '17)*. Article 2, 6 pages.
[5] Flare [n. d.]. Flare. https://medium.com/@BitfuryGroup/the-bitfury-group-releases-\ white-paper-flare-an-approach-to-routing-in-lightning-network-8bc263dcdc92.
[6] L.R. Ford and D.R. Fulkerson. 1954. Maximal flow through a network. *Canadian Journal of Mathematics* 8 (1954).
[7] A.V. Goldberg and R. E. Tarjan. 1988. A new approach to the maximum flow problem. *J. of ACM* 35 (1988), 921–940.
[8] A. M. Kakhki, C. Kliman-Silver, and A. Mislove. 2013. Iolaus: securing online content rating systems. In *International World Wide Web Conference, WWW*. 919–930.
[9] Linux Foundation. 2015. Hyperledger Project. https://{www.hyperledger.org}
[10] G. Malavolta, P. Moreno-Sanchez, A. Kate, and M. Maffei. 2017. SilentWhispers: Enforcing Security and Privacy in Decentralized Credit Networks. In *Annual Network and Distributed System Security Symposium, NDSS*.
[11] G. Malavolta, P. Moreno-Sanchez, A. Kate, M. Maffei, and S. Ravi. 2017. Concurrency and Privacy with Payment-Channel Networks. In *Proceedings ACM SIGSAC Conference on Computer and Communications Security, CCS*. 455–471.
[12] A. Mislove, A. Post, P. Druschel, and P. K. Gummadi. 2008. Ostra: Leveraging Trust to Thwart Unwanted Communication. In *Proceedings of the USENIX Symposium on Networked Systems Design & Implementation, NSDI*. 15–30.
[13] A. Mohaisen, N. Hopper, and Y. Kim. 2011. Keep your friends close: Incorporating trust into social network-based Sybil defenses. In *IEEE International Conference on Computer Communications, INFOCOM*. 1943–1951.
[14] P. Moreno-Sanchez, A. Kate, M. Maffei, and K. Pecina. 2015. Privacy Preserving Payments in Credit Networks: Enabling trust with privacy in online marketplaces. In *Annual Network and Distributed System Security Symposium, NDSS*.
[15] P. Moreno-Sanchez, N. Modi, R. Songhela, A. Kate, and S. Fahmy. 2018. Mind Your Credit: Assessing the Health of the Ripple Credit Network. In *Proceedings of the World Wide Web Conference on World Wide Web, WWW*. 329–338.
[16] NSNAM.org. 2008. Network Simulator 3. https://www.nsnam.org/

[17] A. Post, V. Shah, and A. Mislove. 2011. Bazaar: Strengthening User Reputations in Online Marketplaces. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation, NSDI.*

[18] Ripple [n. d.]. Ripple website. https://ripple.com.

[19] Ripple [n. d.]. Several global banks join RIpple's growing network. https://ripple.com/insights/several-global-banks-join-ripples-growing-network/.

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

[21] Stellar [n. d.]. Stellar website. https://stellar.org.

[22] The OpenSSL Project. 1998. OpenSSL: The Open Source toolkit for SSL/TLS. https://{www.openssl.org}

[23] P. F. Tsuchiya. 1988. The Landmark Hierarchy: A New Hierarchy for Routing in Very Large Networks. In *Symposium Proceedings on Communications Architectures and Protocols (SIGCOMM '88).* 35–42.

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

12 APPENDICES

A *FIND ROUTE* PHASE ALGORITHMS

We give details of the algorithms called in the steps of Algorithm 1. All the algorithms below will have common inputs, **CI** defined as: **CI** = {Set of all RHs, no. of paths n, security parameter λ, hash function H, public ledger BC, **hopMax**}

Algorithm 6: ***Find Route*** Phase: Sender/Receiver Start

```
Input   : CI (defined above)
Output  : find and findReceive tuples of sender and receiver
Parties : Sender: Alice, Receiver: Bob
1  for $i \in [1..n]$ do
       /* Sender start */
2      begin
3          Alice picks a $x' \leftarrow \{0,1\}^\lambda$, computes txid' $\leftarrow H(x')$,
             finds $\mathbf{currMax}_A$,
4          reserves the amount, sets $\mathbf{currMax}_s \leftarrow \mathbf{currMax}_A$.
5          Constructs tuple:
             find(txid', $(VK_C, VK_D)$, reserve($\mathbf{currMax}_s$), hopMax, $C_A$),
             where $C_A = E_{PK_D}(K_{AD}, y \leftarrow \{0,1\}^\lambda, \text{tS})$, $K_{AD}$ is a
             shared symmetric key between Alice and Denise, $y$ is
             a nonce, $PK_D$ is Public Key of Denise and tS is
             timestamp for tuple. The find tuple is sent to all of
             Alice's neighbors.
6      end
       /* Receiver start */
7      begin
8          Parallelly, Bob picks a $x''' \leftarrow \{0,1\}^\lambda$, computes
             txid''' $\leftarrow H(x''')$, finds and reserves $\mathbf{currMax}_B$, sets
             $\mathbf{currMax}_r \leftarrow \mathbf{currMax}_B$.
9          Constructs tuple:
             findReceive(txid''', $VK_D$, reserve($\mathbf{currMax}_r$), hopMax, $C_B$)
10         where $C_B = E_{PK_D}(K_{BD}, y' \leftarrow \{0,1\}^\lambda, \text{tS})$; $K_{BD}$ is
             shared symmetric between Bob and Denise. The
             findReceive tuple is sent to all of Bob's neighbors.
11     end
12 end
```

Algorithm 6: In the ***Find Route*** phase, we have three separate transaction ids, **txid′**, **txid″**, **txid‴** for finding routes on seg_{AC}, seg_{CD}, and seg_{DB} respectively. Later, in the ***Hold*** and ***Pay*** phases, the three transaction ids will coalesce into a single id, **txid**. This is done to hide **txid** from nodes that may not be on the actual path during the ***Hold*** and ***Pay*** phases, but might receive a *find* broadcast message in the ***Find Route*** phase.

For finding a route to their RHs, Alice (Line 2-6) and Bob (Line 7-11) broadcast *find* and *findReceive* tuples containing the payment information to all their neighbors (with whom they have links), who will then forward the tuples to their neighbors, and so on, until the tuples reach the respective RHs. Bob's outgoing tuple is called *findReceive*, to distinguish it from *find*, since the payment will be credited to Bob via his incoming links, as opposed to Alice, whose payment will get debited via her outgoing links. The distance the *find* and *findReceive* tuples have to travel is determined by a system-wide parameter, **hopMax**, which is set by the originator of a tuple. If **hopMax** is underestimated by an originator (i.e., the *find*, *findReceive* tuples cannot reach their destination since **hopMax** is too low), then the originator will retry after a reasonable time interval. The *find* and *findReceive* tuples consist of 5 variables: the transaction id for payment of α_i, public keys of RHs, the current running value of the max. available credit on the path, the maximum hop count, and encrypted info by Alice or Bob for their RHs. This encrypted info includes a unique symmetric key generated by the sender, a challenge nonce (y) and timestamp of the generated tuple by the original sender, encrypted by the public key of the target RH. $\mathbf{currMax}_s$ denotes a running value of the maximum credit available at each intermediate node's outgoing links, on the seg_{AC}, seg_{CD} paths.

Algorithm 7: Starting from Alice, each intermediate node j will *reserve*, or subtract $\mathbf{currMax}_s$ from its max. available credit, $\mathbf{currMax}_j$, if $\mathbf{currMax}_j > \mathbf{currMax}_s$ (Line 3-9), else j will reserve its max. available credit for this transaction. It will then set $\mathbf{currMax}_s$ to be the minimum of $(\mathbf{currMax}_s, \mathbf{currMax}_j)$. Similarly, $\mathbf{currMax}_r$ is the running value of the available max. credit at each intermediate node on the Bob-Denise path (Line 13-18). In case the actual payment does not come before the timeout each node in the path will add the reserved amount back to its credit links.

Algorithm 8, Algorithm 9: Once Charlie receives the find tuple, he picks a random **txid″** and modifies find tuple received from Alice before forwarding it towards Denise to construct seg_{CD} subpath (Alg 8 Line 2). Denise receives the *find* tuple from Alice and constructs a *findReply* tuple which consists of $\mathbf{currMax}_s$ representing the max. credit available on seg_{AC} and seg_{CD} along with some encrypted information for Alice. Denise forwards this tuple along seg_{CD} back towards Charlie along the single path that reached Denise from Charlie represented by nodes with **txid″** (Alg 8 Line 3-12). Denise also receives *findReceive* tuple from Bob and will reply with the maximum available credit on seg_{BD} along with encrypted information for Bob in a *findReply* tuple (Alg 9 Line 6-12). The *findReply* tuples will be sent only to the first neighbor from whom the corresponding *find*/*findReceive* tuple for that transaction id was received. A *findReply* tuple consists of 4 variables: the transaction id for payment of α_i, public keys of RHs, the current running value of $\mathbf{currMax}_s$ or $\mathbf{currMax}_r$, and some information encrypted by RHs for the sender or receiver.

Algorithm 7: ***Find Route*** Phase: Path Construction

```
Input   : CI
Output  : Sender-helper path, receiver-helper path
Parties : Sender: Alice, Receiver: Bob
1  for $i \in [1..n]$ do
       /* Sender-helper path construction */
2      for neighbors $j \in [1..\mathbf{M}]$ in credit path between Alice-Charlie do
3          if ($\mathbf{hopMax} = 0$) then
4              do nothing
5          end
6          else
7              Reserve $\mathbf{currMax}_j$ by $\min(\mathbf{currMax}_s, \mathbf{currMax}_j)$,
                 set $\mathbf{currMax}_s = \min(\mathbf{currMax}_s, \mathbf{currMax}_j)$.
8              Construct tuple:
                 $find(\mathbf{txid}', (VK_C, VK_D), reserve(\mathbf{currMax}_s), (\mathbf{hopMax}-1), C_A)$.
9              Send tuple to all neighbors to whom $j$ has an
                 outgoing credit link.
10         end
11     end
       /* Receiver-helper path construction, done in
          parallel with sender-helper path
          construction */
12     for neighbors $j \in [1..\mathbf{M}]$ in credit path between Bob-Denise
         do
13         if ($\mathbf{hopMax} = 0$) then
14             do nothing
15         end
16         else
17             Reserve $\mathbf{currMax}_j$ by $\min(\mathbf{currMax}_r, \mathbf{currMax}_j)$,
                 set $\mathbf{currMax}_r = \min(\mathbf{currMax}_r, \mathbf{currMax}_j)$.
                 Construct tuple:
                 $findReceive(\mathbf{txid}''', VK_D, reserve(\mathbf{currMax}_r), \mathbf{hopMax}-1, C_B)$.
18             Send tuple to all neighbors to whom $j$ has an
                 incoming link.
19         end
20     end
21 end
```

B SECURITY ANALYSIS

(1) $\mathcal{F}_{\mathsf{FindRoute}}$: The ideal functionality, $\mathcal{F}_{\mathsf{FindRoute}}$, is given in Figure 5. The goal of $\mathcal{F}_{\mathsf{FindRoute}}$ is to find paths between Alice and Bob, and compute the maximum transactable amounts along the paths.
In the first step, the sender Alice, sends the ideal functionality the amount to be transacted, α, the receiver's identity, and the number of shares n of α. She also sends $\mathcal{F}_{\mathsf{FindRoute}}$ the identities of the RHs that Alice and Bob have agreed to use for each share. In Step 2, for each share, $\mathcal{F}_{\mathsf{FindRoute}}$ starts an instance of BFS rooted at Alice, and sends each node the transaction id, the id of its parent, and the id of the target RH for this segment, Charlie. The node replies

Algorithm 8: ***Find Route*** Phase: Helpers' Max. Value Computation

```
Input   : CI
Output  : Max. transaction value computed by $RH$s
Parties : Sender: Alice, Receiver: Bob
1  for $i \in [1..n]$ do
       /* Sender-helper max. computation */
2      When Charlie gets the find$(\cdot,\cdot,\cdot,\cdot,\cdot)$ tuple from Alice, he
         does:
         • Pick $x'' \leftarrow \{0,1\}^\lambda$, compute $\mathbf{txid}'' \leftarrow H(x'')$, reserve
           $\mathbf{currMax}_C$ by $\min(\mathbf{currMax}_s, \mathbf{currMax}_C)$, set
           $\mathbf{currMax}_s = \min(\mathbf{currMax}_s, \mathbf{currMax}_C)$.
         • Store tuple $(\mathbf{txid}', \mathbf{txid}'', VK_D, reserve(\mathbf{currMax}_s))$, create
           new tuple:
           $find(\mathbf{txid}'', (VK_C, VK_D), reserve(\mathbf{currMax}_s), \mathbf{hopMax}, C_A)$.
           The find tuple is
           then sent to all Charlie's neighbors.
       for neighbors $j \in [1..\mathbf{M}]$ in path between Charlie-Denise do
           if ($\mathbf{hopMax} = 0$) then
               do nothing.
           end
           else
               Reserve $\mathbf{currMax}_j$ by $\min(\mathbf{currMax}_s, \mathbf{currMax}_j)$,
               set $\mathbf{currMax}_s = \min(\mathbf{currMax}_s, \mathbf{currMax}_j)$.
               Construct tuple:
               $find(\mathbf{txid}'', (VK_C, VK_D), reserve(\mathbf{currMax}_s), \mathbf{hopMax}-1, C_A)$.
               Send tuple to all neighbors.
           end
       end
       /* Max. in path between $RH$s */
3      When Denise gets the find$(\cdot,\cdot,\cdot,\cdot,\cdot)$ tuple from Charlie,
         she retrieves $(K_{AD}, y, \mathrm{tS}) \leftarrow D_{DK_D}(C_A)$.
4      if decryption fails then
5          do nothing.
6      end
7      else
8          Reserve $\mathbf{currMax}_D$ by $\min(\mathbf{currMax}_s, \mathbf{currMax}_D)$, set
             $\mathbf{currMax}_s = \min(\mathbf{currMax}_s, \mathbf{currMax}_D)$.
9          Store tuple $(\mathbf{txid}'', K_{ABD}, y, VK_C, reserve(\mathbf{currMax}_s))$.
10         Construct tuple:
             $findReply(\mathbf{txid}'', (VK_C, VK_D), C_D, (m, \sigma_D))$, where
             $C_D = E_{K_{AD}}(reserve(\mathbf{currMax}_s), y, \mathrm{tS})$,
             $m = reserve(\mathbf{currMax}_s)$,
             $\sigma_D = Sign_{SK_D}(VK_C, reserve(\mathbf{currMax}_s))$. The
             findReply tuple will be
11         forwarded only to those neighbors on the path with
             Charlie, who have
12         used $\mathbf{txid}''$.
13     end
14 end
```

Algorithm 9: ***Find Route*** Phase: Helpers Reply

```
Input   : CI
Output  : $findReply$ tuples of $RH$s
Parties : Sender: Alice, Receiver: Bob
1  for $i \in [1..n]$ do
       /* Sender's $RH$ sending reply */
2      Charlie, on receipt of Denise's $findReply()$ does:
3      Retrieve $\mathbf{txid'}$ stored in same tuple as $\mathbf{txid''}$, sets his copy
         of $\mathbf{currMax}_s$ to be the $\mathbf{currMax}_s$ received from Denise.
4      Compose reply to Alice:
         $findReply(\mathbf{txid'}, VK_C, E_{K_{AD}}(\mathbf{currMax}_s, y', ts), C_A)$.
5      The $findReply$ tuple will be forwarded only to those
         neighbors on the path from Charlie-Alice, who have used
         $\mathbf{txid'}$.
       /* Receiver's $RH$ sending reply */
6      In parallel, Denise, on receiving Bob's message will
         retrieve $(K_{BD}, y', \mathrm{tS}) \leftarrow D_{DK_D}(C_B)$
7      if decryption fails then
8          do nothing.
9      end
10     else
11         Reserve $\mathbf{currMax}_D$ by $\min(\mathbf{currMax}_r, \mathbf{currMax}_D)$, set
             $\mathbf{currMax}_r = \min(\mathbf{currMax}_r, \mathbf{currMax}_D)$.
12         Compose reply to Bob
             $findReply(\mathbf{txid'''}, VK_D, E_{K_{BD}}(\mathbf{currMax}_r, y', ts), C_B)$.
             The $findReply$ tuple will be forwarded only to those
             neighbors on the path from Denise-Bob, who have
             used $\mathbf{txid'''}$.
13     end
14 end
```

with the ids of its children (degree of the node) and the link weights of each connecting edge. Alternatively, the node can reply with $(\mathbf{txid}, \perp, \perp)$, upon which $\mathcal{F}_{\mathsf{FindRoute}}$ will ignore this node and continue visiting the other children of its parent node. In addition to modeling malicious nodes, this also models nodes that get disconnected from the network, or faulty nodes. $\mathcal{F}_{\mathsf{FindRoute}}$ stores a tuple for each visited node, j, $(j_p, j, \mathbf{currMax}_j)$, and also updates the adjacency matrix with the current link weight values.

In Step 3, once the RH, Charlie is reached, $\mathcal{F}_{\mathsf{FindRoute}}$ computes the max. value that can be transacted along the Alice-Charlie segment, which is the minimum of all the stored **currMax** values seen so far, $\mathbf{currMax}_1$, and sets $\alpha_i = \mathbf{currMax}_1$. Note that there could be possibly multiple routes to Charlie, $\mathcal{F}_{\mathsf{FindRoute}}$ will terminate this segment after the first time Charlie is reached. Similarly, $\mathcal{F}_{\mathsf{FindRoute}}$ will find a path between Charlie and Denise, and Denise and Bob, and will find the max. value that can be transacted between them, $\mathbf{currMax}_2$ and $\mathbf{currMax}_3$ (Step 4, 5). Finally, it computes $\alpha_i = min(\mathbf{currMax}_1, \mathbf{currMax}_2, \mathbf{currMax}_3)$, and informs Alice and Bob about α_i (Step 6). Note that any node can give fake ids of its children and cause delays, but $\mathcal{F}_{\mathsf{FindRoute}}$ will terminate once the max. path length, **hopMax** has been reached.

(2) $\mathcal{F}_{\mathsf{Hold}}$: The $\mathcal{F}_{\mathsf{Hold}}$ functionality is given in Figure 6. For each share of the total amount, the goal of the $\mathcal{F}_{\mathsf{Hold}}$ functionality is to create pairwise contracts between neighboring nodes that commit to transacting that share.
In the first step, $\mathcal{F}_{\mathsf{Hold}}$ divides the path into three segments: Alice-Charlie, Charlie-Denise, and Denise-Bob, so that if a hold-failure occurs in any of the segments (e.g., due to unresponsive, or malicious nodes), only that segment will be re-tried upto a threshold number of times. For each share, starting from Alice, the ideal functionality sends each pair of neighboring nodes (j, k), a tuple consisting of the transaction id, the amount α_i, and a contract, $C_{h_{jk}}$, consisting of the current link weight, $lw_{c_{jk}}$ and the future link weight (after the ***Pay*** phase), $lw_{f_{jk}}$ between j and k. In Step 2, each node can choose to either accept or reject the contract (if a node doesn't respond within a time period, $\mathcal{F}_{\mathsf{Hold}}$ assumes its response is $\perp$). If a node along the Alice-Charlie segment, rejects the contract, $\mathcal{F}_{\mathsf{Hold}}$ sends failure messages to all nodes along the segment, writes a $\mathbf{hold} - \mathbf{fail}$ message to the blockchain and terminates. If any node along the Charlie-Denise and/or Denise-Bob segment rejects the contract, $\mathcal{F}_{\mathsf{Hold}}$ will retry the failed segment a fixed number of times, while the nodes in the successful segments will hold on to their contracts. When a node receives a failure message from $\mathcal{F}_{\mathsf{Hold}}$, it will drop its held contracts. If no node rejects the hold contract, $\mathcal{F}_{\mathsf{Hold}}$ stores a tuple $(\mathbf{txid}, \mathsf{write}, C_{h_{jk}})$ for every pair of nodes j, k. When Bob is reached, he can either choose to accept or reject the contract (Step 3). If Bob accepts, all stored tuples are written to the blockchain, and $\mathcal{F}_{\mathsf{Hold}}$ notifies each node along the path individually that the hold operation was successful (Step 4). Note that the contracts are established iteratively in a pairwise manner to enable any node along the path to potentially abort the hold operation.

(3) $\mathcal{F}_{\mathsf{Pay}}$: The $\mathcal{F}_{\mathsf{Pay}}$ functionality is given in Figure 7. For each share, the goal of $\mathcal{F}_{\mathsf{Pay}}$ is to finalize the transaction by subtracting the credit values on links along the path from Alice to Bob, and writing it to the blockchain. $\mathcal{F}_{\mathsf{Pay}}$ works similar to $\mathcal{F}_{\mathsf{Hold}}$, the only difference being that at the successful completion of $\mathcal{F}_{\mathsf{Pay}}$ for all nodes, the matrix stored by $\mathcal{F}_{\mathsf{DCN}}$ will be updated with the new, decremented link weights.

(4) $\mathcal{F}_{\mathsf{BC}}$: The blockchain functionality is given in Figure 8. $\mathcal{F}_{\mathsf{BC}}$ receives messages from $\mathcal{F}_{\mathsf{Hold}}$ and $\mathcal{F}_{\mathsf{Pay}}$. $\mathcal{F}_{\mathsf{BC}}$ writes tuples to the blockchain, and sends a copy of the new block to nodes (as in the other other ideal functionalities, we assume that all communication between $\mathcal{F}_{\mathsf{BC}}$ and nodes take place via secure and authenticated channels). This is done by sending (update, B). The node can either accept the update, or decline (unresponsive, disconnected, or non-cooperating nodes). When a new node joins the network, or a dormant node wishes to update itself, it can request a copy of the full blockchain by sending a read message to $\mathcal{F}_{\mathsf{BC}}$.

B.1 Design Choices for Ideal Functionalities

In this section, we give the motivation and reasoning behind some of our design choices for the ideal functionalities.

$\mathcal{F}_{\mathsf{FindRoute}}$ Ideal Functionality

(1) Sender Alice sends a tuple $(\alpha, \mathsf{ID}_{\mathsf{Bob}}, n)$ to $\mathcal{F}_{\mathsf{FindRoute}}$. For each $i \in [1..n]$, Alice sends a tuple $(\mathsf{ID}_{\mathsf{Charlie}}, \mathsf{ID}_{\mathsf{Denise}})$ to $\mathcal{F}_{\mathsf{FindRoute}}$ (RHs could be different for each share i).

(2) For each $i \in [1..n]$, $\mathcal{F}_{\mathsf{FindRoute}}$ starts an instance of breadth first search (BFS), rooted at Alice. $\mathcal{F}_{\mathsf{FindRoute}}$ will send each node j, a tuple $(\mathbf{txid}, j_p, \mathsf{ID}_{\mathsf{Charlie}})$. Each node replies to $\mathcal{F}_{\mathsf{FindRoute}}$ with a tuple $(\mathbf{txid}, (j_1, \mathbf{currMax}_1), \ldots, (j_d, \mathbf{currMax}_d))$, or it replies with $(\mathbf{txid}, \perp, \perp)$, where d is the degree of j. In the latter case, $\mathcal{F}_{\mathsf{FindRoute}}$ backtracks to the parent node, and ignores this child. For each visited node j, $\mathcal{F}_{\mathsf{FindRoute}}$ stores a tuple $(j_p, j, \mathbf{currMax}_j)$. $\mathcal{F}_{\mathsf{FindRoute}}$ also updates the matrix stored by $\mathcal{F}_{\mathsf{DCN}}$ with the $\mathbf{currMax}$ (link weight) values.

(3) Once Charlie is reached, or the maximum path length, **hopMax**, has been exceeded, $\mathcal{F}_{\mathsf{FindRoute}}$ terminates the BFS for the Alice-Charlie segment. In the former case, $\mathcal{F}_{\mathsf{FindRoute}}$ computes the minimum of all the stored $\mathbf{currMax}$ values so far, to find the max. value that can be transacted along the Alice-Charlie segment, $\mathbf{currMax}_1$, and sets $\alpha_i = \mathbf{currMax}_1$. In the latter case, $\mathcal{F}_{\mathsf{FindRoute}}$ aborts.

(4) Next, $\mathcal{F}_{\mathsf{FindRoute}}$ starts an instance of BFS rooted at Charlie, with the goal of finding a path between Charlie and Denise, which proceeds similar to Step 2. If Charlie replies with $(\mathbf{txid}, \perp, \perp)$, $\mathcal{F}_{\mathsf{FindRoute}}$ will ask Alice to select a new RH in place of Charlie, and will re-compute α_i. Once Denise is reached, or the max. path length, **hopMax**, has been exceeded, $\mathcal{F}_{\mathsf{FindRoute}}$ terminates the BFS for the Charlie-Denise segment. In the former case, $\mathcal{F}_{\mathsf{FindRoute}}$ computes the minimum of the stored $\mathbf{currMax}$ values along the Charlie-Denise segment to compute the max. value that can be transacted between Charlie and Denise, $\mathbf{currMax}_2$. If $\mathbf{currMax}_2 < \alpha_i$, it sets $\alpha_i = \mathbf{currMax}_2$. If the **hopMax** has been exceeded without reaching Denise, $\mathcal{F}_{\mathsf{FindRoute}}$ will retry finding a path between Charlie and Denise a fixed number of times; if all are unsuccessful, $\mathcal{F}_{\mathsf{FindRoute}}$ aborts.

(5) $\mathcal{F}_{\mathsf{FindRoute}}$ then finds a path between Denise and Bob in a similar manner, and computes the max. value that can be transacted along that path, $\mathbf{currMax}_3$. If Denise replies with $(\mathbf{txid}, \perp, \perp)$, $\mathcal{F}_{\mathsf{FindRoute}}$ will ask Alice to select a new RH in place of Denise, and will re-compute α_i. If $\mathbf{currMax}_3 < \alpha_i$, it sets $\alpha_i = \mathbf{currMax}_3$.

(6) $\mathcal{F}_{\mathsf{FindRoute}}$ sends a tuple $(\mathbf{txid}, \alpha_i, \mathsf{ID}_{\mathsf{Charlie}}, \mathsf{ID}_{\mathsf{Denise}})$ to Alice and Bob. They can respond with $(\mathbf{txid}, \mathsf{find}-\mathsf{accept})$ or $(\mathbf{txid}, \perp)$. If $\mathcal{F}_{\mathsf{FindRoute}}$ receives a $(\mathbf{txid}, \perp)$ from either of them, it aborts. Else it sends them both $(\mathbf{txid}, \mathsf{find}-\mathsf{success})$, outputs all stored tuples of the form $(j_p, j, currmax_j)$ on the path from Alice to Bob, and the final value of α_i, and terminates.

Figure 5: Ideal functionality for the *Find Route* phase

$\mathcal{F}_{\mathsf{Hold}}$ Ideal Functionality

(1) $\mathcal{F}_{\mathsf{Hold}}$ will follow the paths constructed by $\mathcal{F}_{\mathsf{FindRoute}}$ between Alice and Bob, and use the α_i values output by $\mathcal{F}_{\mathsf{FindRoute}}$ for each path. For each $i \in [1..n]$ $\mathcal{F}_{\mathsf{Hold}}$ divides the path into 3 segments: Alice-Charlie (seg_{AC}), Charlie-Denise (seg_{CD}), and Denise-Bob (seg_{BD}). $\mathcal{F}_{\mathsf{Hold}}$ constructs and sends to every pair of neighboring nodes j, k, a tuple $(\mathbf{txid}, \alpha_i, C_{h_{jk}} = (lw_{c_{jk}}, lw_{f_{jk}}))$, where j is k's preceding node.

(2) Nodes j, k can send $(\mathbf{txid}, C_{h_{jk}}, \mathsf{accept}-\mathsf{hold})$ or $(\mathbf{txid}, C_{h_{jk}}, \perp)$ to $\mathcal{F}_{\mathsf{Hold}}$. Depending on which segment it is in, $\mathcal{F}_{\mathsf{Hold}}$ responds thus:
   (a) If any node in seg_{AC} replies with $(\mathbf{txid}, C_{h_{jk}}, \perp)$, $\mathcal{F}_{\mathsf{Hold}}$ aborts the transaction by sending $(\mathbf{txid}, \mathsf{hold}-\mathsf{fail})$ to all nodes, writes $(\mathbf{txid}, \mathsf{hold}-\mathsf{fail})$ to the blockchain by calling $\mathcal{F}_{\mathsf{BC}}$, and terminates. When nodes receive the $(\mathbf{txid}, \mathsf{hold}-\mathsf{fail})$ message, they delete their contracts. Else, if nodes j, k accept the hold message, $\mathcal{F}_{\mathsf{Hold}}$ stores a tuple $(\mathbf{txid}, \mathsf{hold}, C_{h_{jk}})$.
   (b) If any node in seg_{CD} or seg_{BD} replies with $(\mathbf{txid}, C_{h_{jk}}, \perp)$, $\mathcal{F}_{\mathsf{Hold}}$ re-tries the transaction only along the failed segment a fixed number of times. If all re-tries fail, $\mathcal{F}_{\mathsf{Hold}}$ deletes stored tuples, sends $(\mathbf{txid}, \mathsf{hold}-\mathsf{fail})$ to all nodes (at which point nodes will drop their held contracts), writes $(\mathbf{txid}, \mathsf{hold}-\mathsf{fail})$ to the blockchain, and terminates. Else, $\mathcal{F}_{\mathsf{Hold}}$ stores a tuple $(\mathbf{txid}, \mathsf{hold}, C_{h_{jk}})$ for all neighboring nodes j, k.

(3) $\mathcal{F}_{\mathsf{Hold}}$ then sends Bob the tuple $(\mathbf{txid}, \mathsf{hold}, \alpha_i)$. If Bob replies with a $(\mathbf{txid}, \perp)$, $\mathcal{F}_{\mathsf{Hold}}$ deletes all stored tuples, sends $(\mathbf{txid}, \mathsf{hold}-\mathsf{fail})$ to all nodes and terminates. If Bob replies with a $(\mathbf{txid}, \mathsf{hold}-\mathsf{success})$, $\mathcal{F}_{\mathsf{Hold}}$ writes $(\mathbf{txid}, \mathsf{hold})$ to the blockchain by calling $\mathcal{F}_{\mathsf{BC}}$ for all neighboring nodes.

(4) If the write call to $\mathcal{F}_{\mathsf{BC}}$ was successful, $\mathcal{F}_{\mathsf{Hold}}$ sends each node in the path between Alice and Bob a tuple $(\mathbf{txid}, \mathsf{hold}-\mathsf{success})$, outputs all stores tuples of the form $(\mathbf{txid}, \mathsf{hold}, C_{h_{jk}})$, and terminates. Else it sends $(\mathbf{txid}, \mathsf{hold}-\mathsf{fail})$ to nodes and terminates.

Figure 6: Ideal functionality for the *Hold* phase

(1) Each node decides on the fly the next node to route to in $\mathcal{F}_{\mathsf{FindRoute}}$. This is because each node should be able to decide if it wants to be part of a transaction or not. $\mathcal{F}_{\mathsf{FindRoute}}$ could, in principle, decrement the link weights in the matrix maintained by $\mathcal{F}_{\mathsf{DCN}}$, but this would take away the deciding power from nodes. Corrupted nodes could redirect $\mathcal{F}_{\mathsf{FindRoute}}$ among themselves, but since $\mathcal{F}_{\mathsf{FindRoute}}$ knows **hopMax**, it will eventually stop, unless Bob is eventually reached, in which case we consider the transaction to be valid.

(2) The functionality of $\mathcal{F}_{\mathsf{Hold}}$ could be implemented by $\mathcal{F}_{\mathsf{Pay}}$, but we model it separately to stay true to the real world protocol.

(3) In all three phases, $\mathcal{F}_{\mathsf{FindRoute}}$, $\mathcal{F}_{\mathsf{Hold}}$, and $\mathcal{F}_{\mathsf{Pay}}$ in the final step, the ideal functionality sends Bob and/or Alice a tuple with the α_i value (among other things). This is to give Alice and Bob one final opportunity to abort the transaction (e.g., if Alice and Bob have changed their mind, do not want to go

$\mathcal{F}_{\text{Pay}}$ Ideal Functionality

(1) For each $i \in [1..n]$, starting from Alice, $\mathcal{F}_{\text{Pay}}$ constructs a tuple $(\textbf{txid}, \alpha_i, C_{p_{jk}} = (lw_{c_{jk}}, lw_{f_{jk}}))$ and sends to every pair of neighboring nodes, j, k. These tuples will corresponds to the hold tuples outputted by $\mathcal{F}_{\text{Hold}}$. $\mathcal{F}_{\text{Pay}}$ will follow the path constructed by $\mathcal{F}_{\text{FindRoute}}$ and $\mathcal{F}_{\text{Hold}}$. The path is divided into segments, similar to the $\mathcal{F}_{\text{Hold}}$ functionality.
(2) Nodes j, k can send $(\textbf{txid}, C_{p_{jk}}, \text{accept} - \text{pay})$ or $(\textbf{txid}, C_{p_{jk}}, \perp)$. Similar to $\mathcal{F}_{\text{Hold}}$, if any node along seg_{AC} replies with $(\textbf{txid}, C_{p_{jk}}, \perp)$, $\mathcal{F}_{\text{Pay}}$ aborts the transaction, initiates a rollback by calling $\mathcal{F}_{\text{BC}}$ to write $(\textbf{txid}, \text{pay} - \text{fail})$ to the blockchain, sends $(\textbf{txid}, \text{pay} - \text{fail})$ to all nodes, and terminates. When a node receives a $(\textbf{txid}, \text{pay} - \text{fail})$ message, it drops its hold and pay contracts. If any nodes along seg_{CD} or seg_{BD} reply with $(\textbf{txid}, C_{p_{jk}}, \perp)$, $\mathcal{F}_{\text{Pay}}$ will retry the transaction a fixed number of times before aborting. Else $\mathcal{F}_{\text{Pay}}$ creates and stores a tuple $(\textbf{txid}, \text{pay}, C_{p_{jk}})$.
(3) When $\mathcal{F}_{\text{Pay}}$ reaches Bob, it will send him a tuple $(\textbf{txid}, \text{pay}, \alpha_i)$. If Bob replies with $(\textbf{txid}, \perp)$, $\mathcal{F}_{\text{Pay}}$ aborts and rolls back the transaction by sending all nodes prior to Bob on the path to Alice $(\textbf{txid}, \text{pay} - \text{fail})$. It also writes $(\textbf{txid}, \text{pay} - \text{fail})$ to the blockchain. Else if Bob replies with a $(\textbf{txid}, \text{pay} - \text{success})$ $\mathcal{F}_{\text{Pay}}$ writes $(\textbf{txid}, \text{pay})$ to the blockchain by calling $\mathcal{F}_{\text{BC}}$. It also updates the matrix stored by $\mathcal{F}_{\text{DCN}}$ with the new, decremented link weights.
(4) If the write call to $\mathcal{F}_{\text{BC}}$ was successful, $\mathcal{F}_{\text{Pay}}$ sends each node in the path $(\textbf{txid}, \text{pay} - \text{success})$, else it sends $(\textbf{txid}, \text{pay} - \text{fail})$.

Figure 7: Ideal functionality for the *Pay* phase

$\mathcal{F}_{\text{BC}}$ Ideal Functionality

(1) $\mathcal{F}_{\text{BC}}$ can receive 4 kinds of write messages: $(\textbf{txid}, \text{hold} - \text{fail})$, $(\textbf{txid}, \text{hold})$, $(\textbf{txid}, \text{pay} - \text{fail})$, $(\textbf{txid}, \text{pay})$ from $\mathcal{F}_{\text{Hold}}$ and $\mathcal{F}_{\text{Pay}}$ respectively. $\mathcal{F}_{\text{BC}}$ writes the tuple to the blockchain and sends a copy of the newest block B to every node in the credit network by sending a tuple (update, B) to all nodes in the credit network.
(2) Each node in the network either replies with (agree, B) or $(\perp, \perp)$. In the former case, the node updates its copy of the blockchain, has the latest copy of the blockchain stored locally, and is as such synced with the blockchain. However, if the reply was $(\perp, \perp)$, the node now has an outdated copy of the blockchain and can to be referred to as an outdated node.
(3) If a new node joins the credit network or an outdated node wants to get synced with the blockchain, it sends a message (read) to $\mathcal{F}_{\text{BC}}$. $\mathcal{F}_{\text{BC}}$ now sends a message (update, B') where B' is the copy of the entire blockchain so the new/outdated node is synced with the blockchain.

Figure 8: Ideal functionality for blockchain

through with the transaction for any reason, or if they get disconnected from the network).

(4) Alice and Bob give $\mathcal{F}_{\text{FindRoute}}$ the number of shares n and the RHs for each share at the beginning, before any paths have been found. This is one possible way to model $\mathcal{F}_{\text{FindRoute}}$, since then $\mathcal{F}_{\text{FindRoute}}$ can let them know how much maximum credit is available along each of the n paths. Alice and Bob can re-run $\mathcal{F}_{\text{FindRoute}}$ if they want to route more than $\alpha_1, \ldots, \alpha_n$. Another possible way to model $\mathcal{F}_{\text{FindRoute}}$ is for Alice and Bob to just give it the total amount α, and let it compute the number of shares and paths. The first way gives Alice and Bob the freedom to pick their RHs per share. Since the set of RHs is public, Alice and Bob can pick n as a function of the total number of RHs.

B.2 Correctness of Ideal Functionalities

In this section, we give an informal overview of why our definition of the ideal functionalities preserves the claimed privacy/security properties. The formal theorem statement and its proof is given in Section B.3.

$\mathcal{F}_{\text{DCN}}$ consists of 4 ideal functionalities $\mathcal{F}_{\text{FindRoute}}, \mathcal{F}_{\text{Hold}}, \mathcal{F}_{\text{Pay}}$ and $\mathcal{F}_{\text{BC}}$. There are certain pieces of information that are unavoidable to reveal; in fact the construction of the ideal functionalities would be unrealistic without revealing this information. All four ideal functionalities reveal the transaction id, **txid**, which is a randomly generated hash digest to all nodes in a path. This will only tell nodes along a path that a certain transaction has traversed them, but cannot tell anything more than that. With this in mind, we recollect that we define value privacy to mean that no node *not* along a path will learn anything about the value transacted for a particular transaction. Now let us consider each of them with respect to the desired security/privacy properties:

(1) $\mathcal{F}_{\text{FindRoute}}$: $\mathcal{F}_{\text{FindRoute}}$ preserves sender/receiver privacy, since it never reveals the sender/receiver identity to any node in the network. Link privacy is preserved, since every node (including sender/receiver) only knows its parent and children.
(2) $\mathcal{F}_{\text{Hold}}, \mathcal{F}_{\text{Pay}}$: The $\mathcal{F}_{\text{Hold}}$ and $\mathcal{F}_{\text{Pay}}$ functionalities do not reveal sender/receiver information to any node in the network. They reveal the value, α_i traversed along a path, and every node knows the weights on the links connecting it to to its parent and children, but nothing more. Every node also doesn't get any information about the network topology, other than its own outgoing and incoming links.
(3) $\mathcal{F}_{\text{BC}}$: When messages get written to the blockchain, only the **txid** and hold – fail, pay – fail, hold, pay messages are written to the blockchain. These messages just tell nodes in the network that some transaction, **txid**, was written to the blockchain, and whether its hold and pay phases went through or not. This doesn't reveal sender/receiver identity, or the value transacted, or any information about network topology.

B.3 Analysis

Proof of Theorem 6.3.

Proof. We need to prove that no environment, $\mathcal{Z}$ can distinguish between the outputs of the ideal-world simulator, $\mathcal{S}$, and the real-world probabilistic polynomial-time (PPT) adversary, $\mathcal{A}$. We consider a simulator, $\mathcal{S}$ that internally runs $\mathcal{A}$, and, with the help of the ideal-world functionality $\mathcal{F}_{\mathsf{DCN}}$, provides whatever inputs $\mathcal{A}$ asks for in a run of the real-world protocol. $\mathcal{A}$ will then run the real-world protocol, possibly corrupting users, and will generate outputs in each phase of the protocol, which are given to $\mathcal{S}$. Finally, $\mathcal{S}$ takes $\mathcal{A}$'s outputs and forwards them to $\mathcal{F}_{\mathsf{DCN}}$ (consisting of $\mathcal{F}_{\mathsf{FindRoute}}, \mathcal{F}_{\mathsf{Hold}}, \mathcal{F}_{\mathsf{Pay}}$), who completes the simulation of the protocol in the ideal-world.

Our goal here is to show that if $\mathcal{A}$ tries to cheat at *any* point in the execution of the real-world protocol, it will result in *both*, the ideal-world and real-world aborting. In other words, it is not possible for $\mathcal{A}$ to cheat in the real-world in such a way that the ideal-world and real-world simulations go through properly. It follows that if $\mathcal{S}$ can exactly mirror the actions of $\mathcal{A}$ in the real-world (successfully completing when $\mathcal{A}$ completes, and aborting when $\mathcal{A}$ tries to cheat, or $\mathcal{A}$ aborts), then no environment $\mathcal{Z}$ can successfully distinguish between the two worlds, except with negligible probability, where the probabilities are taken over the random coins of $\mathcal{A}$.

Let us consider a complete run of the ideal world and real-world protocols consisting of the ***Find Route*** phase, the ***Hold*** phase and the ***Pay*** phase.

- ***Find Route*** **phase**: In the ***Find Route*** phase, $\mathcal{S}$ will generate the set of inputs for $\mathcal{A}$: the set of RHs, the number of paths to be found, n, security parameter λ, hash function H, a public ledger BC, and the max. path length, **hopMax**. Since all of these parameters are public, $\mathcal{S}$ can generate them by itself. $\mathcal{A}$ then picks two RHs, Charlie and Denise, generates random $\mathbf{txid}'$ and $\mathbf{txid}''$ along the Alice-Charlie and Bob-Denise path, constructs the *find* and *findReceive* tuples, and gives them to $\mathcal{S}$. Now, $\mathcal{S}$ needs to simulate this correctly in the ideal-world. $\mathcal{S}$ will pick the same RHs that $\mathcal{A}$ picked, Charlie and Denise, and call $\mathcal{F}_{\mathsf{FindRoute}}$ with parameters ($\mathbf{txid}, \alpha, \mathsf{ID}_{\mathsf{Bob}}, n, \mathsf{ID}_{\mathsf{Charlie}}, \mathsf{ID}_{\mathsf{Denise}}$). Note that $\mathcal{S}$ uses a single **txid**, while forwarding information to $\mathcal{F}_{\mathsf{FindRoute}}$, since in the ideal-world, $\mathcal{F}_{\mathsf{FindRoute}}$, and not the RHs find routes. But $\mathcal{S}$ will record $\mathbf{txid}'$ and $\mathbf{txid}''$, and will use them while giving $\mathcal{A}$ the results of $\mathcal{F}_{\mathsf{FindRoute}}$'s run.
  $\mathcal{F}_{\mathsf{FindRoute}}$ then starts an instance of breadth first search (BFS) rooted at sender Alice, and each node j along the Alice-Charlie path will receive tuples ($\mathbf{txid}, j_p, \mathsf{ID}_{\mathsf{Charlie}}$), where j_p is j's parent. The BFS proceeds as described in Figure 5; once done, $\mathcal{F}_{\mathsf{FindRoute}}$ updates its matrix with link weights. At some point, either Charlie will be reached, or **hopMax** will get exceeded. In the former case, $\mathcal{S}$ will store the value of α_i, and continue the simulation with $\mathcal{F}_{\mathsf{FindRoute}}$ on the Charlie-Denise and Denise-Bob paths. In the latter case (which occurs if $\mathcal{A}$ has corrupted nodes along the path), $\mathcal{F}_{\mathsf{FindRoute}}$ will abort. $\mathcal{S}$ notifies $\mathcal{A}$ of the failure, and terminates the ideal-world execution, following which $\mathcal{A}$ will have to halt the real-world execution, since a path has not been found from Alice to Charlie.
  After Charlie has been reached, $\mathcal{F}_{\mathsf{FindRoute}}$ will continue with the simulation, finding paths from Charlie to Denise and Denise to Bob in a similar way, in serial order. At any point of **hopMax** gets exceeded without reaching the target (i.e., $\mathcal{A}$ corrupts nodes along the way, which forward $\mathcal{F}_{\mathsf{FindRoute}}$ between themselves, or $\mathcal{A}$ aborts), $\mathcal{S}$ notifies $\mathcal{A}$ that the protocol has failed, which results in both, ideal and real worlds aborting. Finally, $\mathcal{S}$ will forward to $\mathcal{A}$ the results of $\mathcal{F}_{\mathsf{FindRoute}}$'s execution: n full paths from Alice to Bob, and $\alpha_1, \ldots, \alpha_n$. Note that before forwarding to $\mathcal{A}$, $\mathcal{S}$ will replace **txid** with $\mathbf{txid}', \mathbf{txid}''$ and $\mathbf{txid}'''$ along the three path segments, respectively.
  One can see that there is no way $\mathcal{A}$ can cause $\mathcal{F}_{\mathsf{FindRoute}}$ to run indefinitely, or find paths to wrong users (e.g., instead of Bob, the credit gets routed to $\mathcal{A}$). This ensures integrity of the transaction. Also, corrupted nodes not directly on a path, will not be given any information by $\mathcal{F}_{\mathsf{FindRoute}}$, and hence will not know the value transacted through them. Recollect that in our adversary model, it is unavoidable that nodes sitting on a path will know the value transacted through them. Finally, unless $\mathcal{A}$ corrupts the next-hop neighbor of either sender or receiver, $\mathcal{A}$ is not given any information about the identities of Alice and Bob by $\mathcal{S}$, i.e., in $\mathcal{F}_{\mathsf{FindRoute}}$, every node only knows its parent and children, and has no information about other nodes.
  Hence, $\mathcal{S}$ exactly mirrors the actions of $\mathcal{A}$ in the ideal-world, aborts at any attempt to cheat by $\mathcal{A}$. It follows that no environment $\mathcal{Z}$ can successfully distinguish between the actions of $\mathcal{A}$ in the real-world and $\mathcal{S}$ in the ideal-world.
- ***Hold*** **phase**: In the ***Hold*** phase, $\mathcal{S}$ will begin by providing $\mathcal{A}$ the set of RHs, the number of paths, n, and the amount to be transacted along each path, $\alpha_1, \ldots, \alpha_n$, hash function H, public blockchain, BC. The main point to notice here is that we are giving the adversary all amounts along all paths; this is because we assume a strong adversary, where $\mathcal{A}$ can corrupt nodes along *all* paths between Alice and Bob, in which case $\mathcal{A}$ will know all the α_i values. $\mathcal{A}$ starts the execution by re-using its $\mathbf{txid}^{prime}, \mathbf{txid}'', \mathbf{txid}'''$ and keys from the ***Find Route*** phase. It constructs the *hold* tuple for Alice, which is forwarded to $\mathcal{S}$. On receiving the *hold* tuple, $\mathcal{S}$ constructs a tuple ($\mathbf{txid}, \alpha_i, H_{jk} = (cw_{jk}, fw_{jk})$), where j is k's preceding node. These tuples are sent starting from Alice's neighbor up until Bob. $\mathcal{S}$ re-uses the **txid** it stored from the find phase. An interesting point is where does $\mathcal{S}$ get (cw_{jk}, fw_{jk}) from? $\mathcal{S}$ contacts $\mathcal{F}_{\mathsf{DCN}}$, which maintains the matrix of link weights, and gets $cw_{jk}$from the matrix; $fw_{jk}$is set to be $cw_{jk} - \alpha_i$, which can be locally computed by $\mathcal{S}$. Now, $\mathcal{S}$ forwards the tuples to $\mathcal{F}_{\mathsf{Hold}}$, who sends the tuples starting from Alice's neighbor, up until Bob is reached. The ideal-world simulation proceeds as described in Figure 6. If any node along the path replies with a ($\mathbf{txid}, H_{jk}, \perp$) (which means $\mathcal{A}$ has corrupted the node), $\mathcal{F}_{\mathsf{Hold}}$ aborts the transaction (after a number of re-tries, if applicable), informs $\mathcal{A}$ of the failure, calls $\mathcal{F}_{\mathsf{BC}}$ and writes ($\mathbf{txid}$, hold − fail) to the blockchain. This will force $\mathcal{A}$ to abort the real-world protocol. If Bob is successfully reached, accepts the *hold* contract, and the final write call to $\mathcal{F}_{\mathsf{BC}}$ is successful, $\mathcal{S}$ sends $\mathcal{A}$ the tuple ($\mathbf{txid}, \alpha_i, H_{jk}$).

It follows that if $\mathcal{A}$ causes any nodes along the path to be unresponsive, does not follow the route created by $\mathcal{F}_{\mathsf{FindRoute}}$, or otherwise tries to route $\mathcal{F}_{\mathsf{Hold}}$ in a circular fashion among corrupt nodes, $\mathcal{S}$ will terminate the ideal-world simulation, which will force $\mathcal{A}$ to abort the real-world execution. Since the ***Hold*** phase has failed, $\mathcal{A}$ cannot continue further with the ***Pay*** phase. We note that since we are considering a strong adversary, who corrupts nodes along *all* paths, $\mathcal{A}$ will know the values of $\alpha_1, \dots, \alpha_n$. In practice, this might rarely be the case, and $\mathcal{A}$ will only know the values transmitted along those paths on which it has planted corrupted nodes.

- ***Pay*** **phase**: The simulation works very similar to the hold phase.

□