# 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 [19] and Stellar [23], 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, 12 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, 26]. 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 credits directly with each other, if there exists a direct trust relationship between them, or via a path between them through network peers, built on peer-wise credit relationships.

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 Ripple payment settlement network [19].

Credit networks have found use in several applications, such as designing and securing social networks [13], Sybil tolerant networks [26], and content rating systems [8]. Popular blockchain-based payment settlement networks (e.g., Ripple [19], Stellar [23]) use credit networks as underlying infrastructure to represent credit between users. Other examples being TrustDavis [3], Bazaar [18], and Ostra [12]. Furthermore, traditional banking systems have begun integrating blockchain-based payment networks such as Ripple into services [21]–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 network pathi with sufficient credit. Once a payment gets routed from a sender to a receiver, all edges along the path get decremented by the transmitted amount.

Both centralized and decentralized credit networks currently exist. In the centralized version [12, 18], 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, 19, 23], 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.

**Challenges**: For broad-based acceptance and use, any credit network has to handle the following three major challenges:


*CODASPY '19, March 25–27, 2019, Richardson, TX, USA*

ACM ISBN 978-1-4503-6099-9/19/03…$15.00
https://doi.org/10.1145/3292006.3300034

**(a)** *Concurrency*: In a credit network, several concurrent transactions could occur (e.g., Ripple's XRP processes up to 1500 transactions per second [20]), 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 – either all credit links on the path get decremented, 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. Users can choose to split their credit requests across multiple paths 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.

In essence, we propose an alternative to proposed landmarks-based routing and DCN maintenance techniques [10, 22, 25], 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 among 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 the 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 to large, dynamic 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 between two users to all nodes in the neighborhood, and works only for Bitcoin transactions. Malavolta *et al.* [11] proposed 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, 26]. In the distributed setting, mechanisms using *landmark routing* [25] 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, 22].

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.* [22] 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 [22], (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 truly on-demand at the start of a transaction.

While [10, 22] 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 [19, 23].

## 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 $\gamma$ from node $j$ to node $k$ signifies that $k$ has extended $\gamma$ 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. 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.

**Table 1: Notations**

| Variable | Definition |
|---|---|
| $\lambda$ | Security parameter |
| RH | Routing helper |
| $\alpha = \{\alpha_1, \ldots, \alpha_n\}$ | $\alpha$ is the total credit amount; each $\alpha_i$ is a fraction of $\alpha$. |
| **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 transaction timeout |
| $\mathbf{currMax}_i$ | Max. link weight of user $i$ |
| $\mathbf{currMax}_{seg_{jk}}$ | Max. link weight along $seg_{jk}$ |
| $cw_{jk}$ | Current link weight between nodes $j$, $k$ |
| $fw_{jk}$ | Future link weight between nodes $j$, $k$ |
| $uw_{jk}$ | Updated link weight between nodes $j$, $k$ |
| $H_{jk}$ | Hold contract between Node $j$, $k$ |
| $P_{jk}$ | Pay contract between Node $j$, $k$ |
| $BC.read()$ $BC.write()$ | 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 $j$ |
| $sk_j, vk_j$ | Long-term signing/verification key-pair of node $j$ |
| $PK_j, DK_j$ | Encryption/decryption key-pair of node $j$ |
| $C_i$ | Ciphertext produced by user $i$ |
| $\sigma$ | Signature |

**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, it is broadcasted to all subscribed nodes. When a node needs to write a message, *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 blockchain will be used for proof of transactions (and misbehavior); any adjudication and punitive enforcement of misbehavior is out of scope of this work. Consensus protocols used by the underlying blockchain do not affect BlAnC, hence are not discussed here. BlAnC is designed for decentralized anonymity, using a database for storing credit link weights will be more efficient but it may leak 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 mutually-agreed upon link weights(credits). They then double sign(each also has the other's signature) the agreed credit values, and store 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 verifcation key 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 long-term verification key of each RH in the system is known to all users, along with the long-term public key of the RH (used for encrypted communication to 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 her adjacent credit links.

*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 (credit is transferred concurrently along multiple paths for 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 by their honest neighbors from being in the credit paths.

*Integrity/Rollback*: If a transaction does not go through successfully (after multiple retries), every credit link 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.

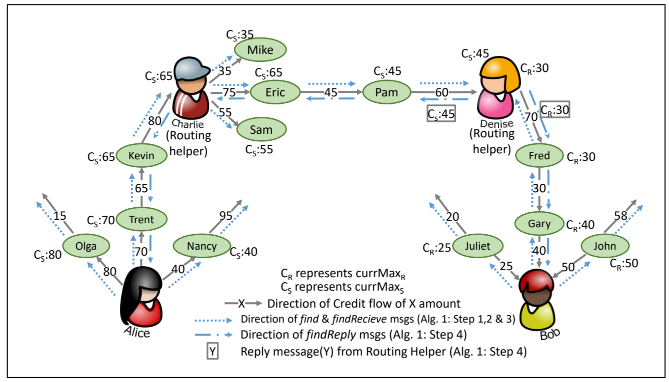


**Figure 1: *Find Route* Phase: Alice and Bob use Charlie and Denise as** RH**s for one of the transaction-split $\alpha_i$.**

## 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 $\alpha$, 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, $\alpha_i$ (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. In this phase, the route between Alice and Bob is segmented at the two RHs: $seg_{AC}$, $seg_{CD}$, and $seg_{DB}$, representing segments between Alice and Charlie, Charlie and Dennis, and Dennis and Bob respectively, with corresponding transaction ids **txid′**, **txid′′**, and **txid′′′**.

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, respectively, and drop all later duplicate messages. For readability, only a single path ($seg_{AC}$, $seg_{CD}$, $seg_{DB}$) between Alice and Bob is shown. Each node stores the mapping of the incoming, outgoing links, and transaction ID per transaction. This info is stored until transaction completes/aborts.

In case the *find* or *findReceive* did not reach the intended RH, 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 $C_s = 45$ ($min(max(seg_{AC}), max(seg_{CD}))$ $\Rightarrow min(65, 45)$), and forwards a *findReply* message to Charlie, who

**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

**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 maximum 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, $\alpha_i = min(\mathbf{currMax}_s, \mathbf{currMax}_r)$.

**end**

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 $\alpha'$.

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 presents the algorithm (see Table 1 for notations). The steps are self-explanatory. The other algorithms invoked within Algorithm 1 are in the Appendix A.

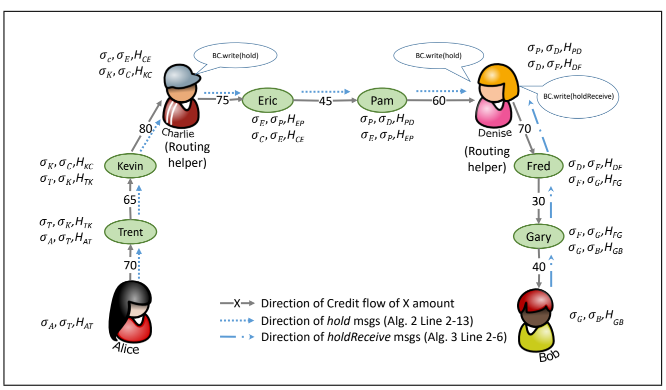


**Figure 2:** ***Hold*** **Phase: Between Alice and Denise on segments** $seg_{AC}$, $seg_{CD}$ **and between Bob and Denise on segment** $seg_{DB}$.

## 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

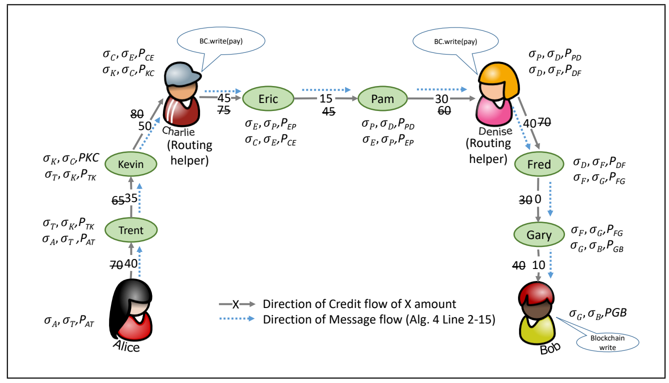


**Figure 3:** ***Pay*** **Phase: Alice creates a** ***pay*** **tuple with** $\alpha_i = 30$**, and forwards it on the Alice-Bob path (in the figure,** $\alpha_i$ **is contained in each** $P_{jk}$ **contract, for nodes** $j, k$**). Each RH writes a message to the BC whenever they receive the** ***pay***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

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, $\alpha_i$ 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 pair(e.g. $j, k$) on $seg_{DB}$ writes the corresponding contract, and their signatures on the contract $(\sigma_j, \sigma_k)$ into their 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.

Alice and Bob pick a pre-image, $x \leftarrow \{0, 1\}^\lambda$ out-of-band, and compute $\mathbf{txid} \leftarrow H(x)$ (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 in turn forwards it to Denise on $seg_{CD}$ (Line 3 in Algorithm 2). The *hold* messages follow the path used by $\mathbf{txid'}$ on $seg_{AC}$, and $\mathbf{txid''}$ 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, $\mathbf{txid}$.

Similarly, Bob will send a *holdReceive* message on $seg_{DB}$, which will follow the path with $\mathbf{txid'''}$ and create pairwise contracts for

**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, **txid′**, **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

**for** $\alpha_i, i \in [1..n]$ **do**

  `/* Hold on sub-path from Alice to Charlie */`

  **begin**

    Alice picks token, $x \leftarrow \{0,1\}^{\lambda}$, **txid** $\leftarrow H(x)$, shares token, $x$, **txid** with Bob; constructs tuple: *hold*(**txid′**||**txid**||$\alpha_i$||$C_A$||tD), where, $C_A = E_{K_{AD}}$(token||$vk_C$||**txid**||tS); sends *hold* to neighbor on path **txid′**.

    **for** *each pair of consecutive nodes* $j, k \in [1..\mathbf{M}]$ *on path* **txid′ do**

      When $k$ receives *hold*(**txid′**, **txid**, $\alpha_i$, $C_A$) from $j$, then $k$ runs MultiSig($j||SK_j||VK_j||k||SK_k||VK_k||$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 **txid′** to **txid**.

    **end**

    Charlie, on receiving *hold*, calls MultiSig(), and updates **txid**, writes a signed message to BC using $BC.write((vk_C||\mathbf{txid}||hold)||Sign_{sk_C}(vk_C||\mathbf{txid}||hold))$.

  **end**

  `/* Hold on sub-path from Charlie to Denise */`

  **begin**

    Charlie updates the tuple to *hold*(**txid″**||**txid**||$\alpha_i$||$C_A$||tD), sends it to neighbor on $seg_{CD}$ with **txid″**.

    The intermediate nodes follow the same procedure as those on $seg_{AC}$, except with **txid″** instead of **txid′** (details omitted due to space constraints).

    Denise, on receiving *hold* tuple, calls MultiSig(), updates **txid**, writes a signed message to BC by calling $BC.write((vk_D||\mathbf{txid}||hold)||Sign_{sk_D}(vk_D||\mathbf{txid}||hold))$.

  **end**

**end**

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 or a signed *holdReceive* message on the BC corresponding to **txid** from their target RH. After the ***Hold*** phase, the three different paths from the ***Find Route*** phase coalesce into a single path marked by **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. ($seg_{AC}$, $seg_{CD}$ and $seg_{DB}$) Finally, Denise writes a signed message to BC by calling $BC.write((vk_D||\mathbf{txid}||holdReceive)||$ $Sign_{sk_D}(vk_D||\mathbf{txid}||holdReceive))$.

6   **end**

7 **end**

## 5.3 *Pay* Phase

At the end of the ***Hold*** phase, all nodes on the path from Alice to Bob would have committed $\alpha_i$ 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**, $\alpha_i$, 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 its neighbors on the path and changes its corresponding 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 timeout value tD; nodes wait for tD time for the target RH to write the *pay* message to the BC, or abort

## 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 no. 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 consensus algorithm synchronizes it across the network. Our DCN can be deployed on any existing BC as long as it supports individual nodes writing signed messages to the BC as opposed to writing transactions, and has 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 wait for the transaction to be retried and completed in the other 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. The offending node's privacy in the network is still maintained and only its immediate neighbors identify it. 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 $BC.write(\textbf{txid}||H_{jk}||P_{jk})$ if they had a *pay* contract or just $BC.write(H_{jk})$ if they did not receive a *pay* message from the 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. All nodes on path should delete their *hold* contracts and revert back to previous link weights if they had any *pay* contracts associated with transaction **txid**. Rollback won't affect other concurrent transactions; no topology-related info is written to BC.

**Algorithm 4: *Pay* Phase**

**Input** : Set of RHs, total amount $\alpha = \alpha_1, \ldots, \alpha_n$, $\lambda$, 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

**for** $\alpha_i, i \in [1..n]$ **do**
  /* Pay on sub-path from Alice to Charlie */
  **begin**
    Alice constructs tuple *pay*(**txid**$||\alpha_i||$tD), sends *pay* tuple to next neighbor on path **txid**
    **for** *each pair of consecutive nodes* $j, k \in [1..\mathbf{M}]$ *in* $seg_{AC}$ *which have* **txid** **do**
      When $k$ receives *pay*(**txid**$||\alpha_i||$tD) from $j$, $k$ runs MultiSig($j \parallel SK_j \parallel VK_j \parallel k \parallel SK_k \parallel VK_k \parallel$ tS$||cw_{jk}||uw_{jk}$) (see Algorithm 5). Nodes $j, k$ each locally store ($\sigma_j, \sigma_k$, ($P_{jk}$= contract)).
    **end**
    On receiving *pay* tuple, after calling MultiSig(), Charlie writes a signed message to BC by calling $BC.write((vk_C||\textbf{txid}||pay)||\text{Sign}_{SK_C}(vk_C||\textbf{txid}||pay))$.
  **end**
  /* Pay on sub-path from Charlie to Denise */
  **begin**
    Charlie forwards *pay* tuple to the next neighbor on **txid** path towards Denise.
    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 $BC.write((vk_D||\textbf{txid}||pay)||\text{Sign}_{SK_D}(vk_D||\textbf{txid}||pay))$. Denise forwards *pay* tuple to the next neighbor on **txid** path towards Bob.
  **end**
  /* Pay on sub-path from Denise to Bob */
  **begin**
    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 $BC.write(\textbf{txid}||x||\textbf{success})$.
  **end**
**end**

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, i.e., no timeouts in a transaction, 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 sixth message is the **success** message written by Bob to the BC. The only information gleaned from BC is that a certain RH pair was involved in a transaction of unknown amount. 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 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 $k$

$j$ sends $\sigma_j \leftarrow \mathrm{Sign}_{SK_j}$(contract = $(cw_{jk}, \gamma)$, **txid**, tS) to $k$
$k$ sends $\sigma_k \leftarrow \mathrm{Sign}_{SK_k}$(contract = $(cw_{jk}, \gamma)$, **txid**, tS) to $j$
**if** $\mathit{Verify}_{VK_k}(\text{contract}||\sigma_k) \stackrel{?}{=} 1$ **then**
  $j$ stores ($\sigma_j||\sigma_k||$contract)
**end**
**if** $\mathit{Verify}_{VK_j}(\text{contract}||\sigma_j) \stackrel{?}{=} 1$ **then**
  $k$ stores ($\sigma_j||\sigma_k||$contract)
**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 the misbehaving RH 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, $\pi$ to be proved secure runs, and the ideal-world, where the entire protocol, $\phi$ 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:

*Definition 6.1.* (UC-emulation [1]) Let $\pi$ and $\phi$ be probabilistic polynomial-time (PPT) protocols. We say that $\pi$ UC-emulates $\phi$ 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

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

*Definition 6.2.* (UC-realization [1]) Let $\mathcal{F}$ be an ideal functionality and let $\pi$ be a protocol. We say that $\pi$ UC-realizes $\mathcal{F}$ if $\pi$ 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. Due to space constraints we have given 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 in the full version of the paper [17].

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.

**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

**Table 2: Asymptotic Complexities: $n$ denotes the number of fractions of a payment (in Ripple [19], 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 [19], the max. path length is 10).**

| Phases | Time | Space | Messages | | |
|---|---|---|---|---|---|
| | | | Regular users | Charlie (RH) | Denise (RH) |
| ***Find Route*** 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 | $O(k \cdot n)$ | $O(k \cdot n)$ | $O(k \cdot n)$ | $O(n)$ | $O(n)$ |

**Table 3: $n$ is the number of fractions, 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 Route*** 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 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 Route*** 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 |

*hold* and *pay* contracts to the BC, exposing their pseudonyms. All involved nodes need to establish new pseudonyms with their neighbors. The time complexity of this step is $O(k \cdot d)$, 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 [24]. 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, since a wireless connection is representative of a majority of users. Given that the Internet's diameter is around 18, in our simulation setup, the sender and receiver are 15 hops apart, on a path passing through the two RH. 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. 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 and 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. The cryptographic operations in the ***Hold*** phase and ***Pay*** phases contribute make the bulk of the cryptographic time delay, that is, ~ 37 *ms* as opposed to 4 *ms* in the ***Find Route*** phase.

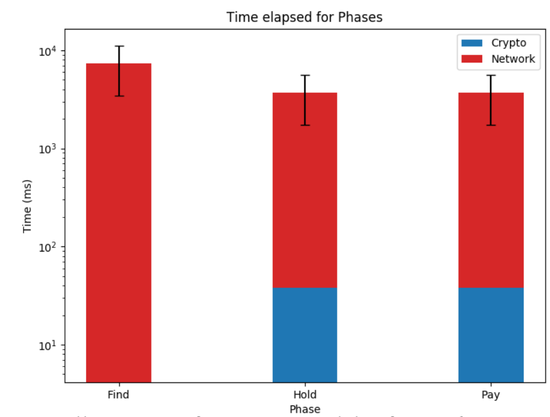


**Figure 4: Illustration of average time delay for *Find Route* , *Hold* and *Pay* phases during a** BlAnC **transaction due to network and cryptographic operations.**

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 $\alpha_i$ 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 $\alpha_i$ on a path of length 10 *hops* (BlAnC is three orders of magnitude faster). The significant time delay (in the ***Find Route*** phase) is 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.

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, 19]. This is a one-time cost, which, will be amortized over time by having the sender/receiver store information about the paths to their respective RHs, 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 to choose $d'$ of its neighbors at random ($d' < d$), and send *find* tuples only to them.

Each node in the credit network that is involved in a transaction, could keep track of the identity of the RH, maximum link weight available to 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 [22] 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 use 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.

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 make sure all the payments and links are settled for past transactions. At this point, the BC can be compacted by replacing all but the last few active blocks (blocks containing active transactions) with a single compacted block that contains the hash of the chain upto that point. 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 authors 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. , 147–156 pages.
[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://bitfury.com/content/downloads/whitepaper_flare_an_approach_to_routing_in_lightning_network_7_7_2016.pdf.
[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] G. Panwar, S. Misra, and R. Vishwanathan. 2019. BlAnC. https://eprint.iacr.org/2019/014
[18] 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*.
[19] Ripple [n. d.]. Ripple website. https://ripple.com.
[20] Ripple [n. d.]. Ripple XRP charts. https://ripple.com/xrp/.
[21] Ripple [n. d.]. Several global banks join RIpple's growing network. https://ripple.com/insights/several-global-banks-join-ripples-growing-network/.
[22] 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.*
[23] Stellar [n. d.]. Stellar website. https://stellar.org.
[24] The OpenSSL Project. 1998. OpenSSL: The Open Source toolkit for SSL/TLS. https://{www.openssl.org}
[25] 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.
[26] 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.

# A *FIND ROUTE* PHASE ALGORITHMS

We give details of the algorithms called in the steps of the ***Find Route*** Algorithm 1. All the algorithms below have common inputs, **CI** defined as: **CI** = {Set of all RHs, no. of paths $n$, security parameter $\lambda$, 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

**for** $i \in [1..n]$ **do**
  /* Sender start */
  **begin**
    Alice picks a $x' \leftarrow \{0,1\}^{\lambda}$, computes $\textbf{txid}' \leftarrow H(x')$, finds $\textbf{currMax}_A$,
    reserves the amount, sets $\textbf{currMax}_s \leftarrow \textbf{currMax}_A$.
    Constructs tuple: *find*($\textbf{txid}'||(vk_C||$"$vk_D)||$*reserve*$(\textbf{currMax}_s)$ $||\textbf{hopMax}||C_A$), 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.
  **end**
  /* Receiver start */
  **begin**
    Parallelly, Bob picks a $x''' \leftarrow \{0,1\}^{\lambda}$, computes $\textbf{txid}''' \leftarrow H(x''')$, finds and reserves $\textbf{currMax}_B$, sets $\textbf{currMax}_r \leftarrow \textbf{currMax}_B$.
    Constructs tuple: *findReceive*($\textbf{txid}'''||vk_D||$*reserve*$(\textbf{currMax}_r)$ $||\textbf{hopMax}||C_B$)
    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.
  **end**
**end**

In Algorithm 6, Alice and Bob generate $\textbf{txid}'$ and $\textbf{txid}'''$ respectively. They also independently generate symmetric encryption keys and challenge nonces for Denise. Alice creates a *find* tuple, containing the $\textbf{txid}'$, $\textbf{currMax}_s$, **hopMax**, identity of both Charlie and Denise, and some encrypted information which is broadcast to all her neighbors. Bob also creates a similar tuple called *findReceive*, containing the $\textbf{txid}''$, $\textbf{currMax}_r$, **hopMax**, identity of Denise, and some encrypted information which is broadcast to all his neighbors. At Alice and Bob $\textbf{currMax}_s$ and $\textbf{currMax}_r$ are respectively set to their outgoing link weights. If the messages timeout, then sender can retry with a higher **hopMax** value.

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

**Input** : **CI**
**Output** : Sender-helper path, receiver-helper path
**Parties** : Sender: Alice, Receiver: Bob

**for** $i \in [1..n]$ **do**
  /* Sender-helper path construction */
  **for** *neighbors* $j \in [1..\textbf{M}]$ *in credit path between Alice-Charlie* **do**
    **if** (**hopMax** = 0) **then**
      do nothing
    **end**
    **else**
      Reserve $\textbf{currMax}_j$ by $\min(\textbf{currMax}_s, \textbf{currMax}_j)$, set $\textbf{currMax}_s = \min(\textbf{currMax}_s, \textbf{currMax}_j)$.
      Construct tuple: *find*($\textbf{txid}'||(vk_C, vk_D)||$*reserve*$(\textbf{currMax}_s)$ $||(\textbf{hopMax} - 1)||C_A$).
      Send tuple to all neighbors to whom $j$ has an outgoing credit link.
    **end**
  **end**
  /* Receiver-helper and sender-helper path construction done concurrently */
  **for** *neighbors* $j \in [1..\textbf{M}]$ *in credit path between Bob-Denise* **do**
    **if** (**hopMax** = 0) **then**
      do nothing
    **end**
    **else**
      Reserve $\textbf{currMax}_j$ by $\min(\textbf{currMax}_r, \textbf{currMax}_j)$, set $\textbf{currMax}_r = \min(\textbf{currMax}_r, \textbf{currMax}_j)$. Construct tuple: *findReceive*($\textbf{txid}'''||vk_D||$*reserve*$(\textbf{currMax}_r)$ $||\textbf{hopMax} - 1||C_B$).
      Send tuple to all neighbors to whom $j$ has an incoming link.
    **end**
  **end**
**end**

In Algorithm 7, each intermediate node on $seg_{AC}$, on receiving the *find* tuple will *reserve* the minimum of $\textbf{currMax}_s$ and it's outgoing link's credit value. The node updates $\textbf{currMax}_s$ (if needed) and also decrements the value of **hopMax** in the *find* tuple before broadcasting it to all its neighbors. Similarly, each intermediate node on $seg_{DB}$, on receiving the *findReceive* tuple will *reserve* the minimum of $\textbf{currMax}_r$ and the available credit on its outgoing link. The procedure is the same as that followed by the intermediate node in $seg_{AC}$, only now the receiver is Denise. If the **hopMax** value was 0 when the node received the *find* or *findReceive* tuple, then the node drops the message.

In Algorithm 8, when Charlie receives the *find* tuple, it creates a new $\textbf{txid}''$, updates the *find* tuple with this new value and broadcasts it towards Denise. Each intermediate node on $seg_{CD}$ from

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

**Input :** CI
**Output :** Max. transaction value computed by *RH*s
**Parties :** Sender: Alice, Receiver: Bob

**for** $i \in [1..n]$ **do**
  /* Sender-helper max. computation */
  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 of 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 */
  When Denise gets the find$(\cdot||\cdot||\cdot||\cdot||\cdot)$ tuple from Charlie, she retrieves $(K_{AD}||y||\text{tS}) \leftarrow D_{DK_D}(C_A)$.
  **if** *decryption fails* **then**
    do nothing.
  **end**
  **else**
    Reserve $\mathbf{currMax}_D$ by $\min(\mathbf{currMax}_s, \mathbf{currMax}_D)$, set $\mathbf{currMax}_s = \min(\mathbf{currMax}_s, \mathbf{currMax}_D)$.
    Store tuple $(\mathbf{txid}''||K_{ABD}||y||vk_C||reserve(\mathbf{currMax}_s))$.
    Construct tuple:
    $findReply(\mathbf{txid}''||(vk_C||vk_D)||C_D||(m||\sigma_D))$, where $C_D = E_{K_{AD}}(reserve(\mathbf{currMax}_s)||y||\text{tS})$, $m = reserve(\mathbf{currMax}_s)$, $\sigma_D = Sign_{sk_D}(vk_C||reserve(\mathbf{currMax}_s))$. The *findReply* tuple will be forwarded to the neighbors on the reverse path with Charlie.
  **end**
**end**

Charlie to Denise follows the same steps as nodes on $seg_{AC}$ mentioned earlier. If the *find* tuple does not reach Denise, and Charlie times out, then Charlie can retry with a higher **hopMax** value. When Denise receives the message, she retrieves the encrypted challenge nonce and the symmetric key. She composes a *findReply* tuple with encrypted information for Alice and forwards it to the node on $seg_{CD}$. This message is forwarded back towards Charlie by each intermediate node.

In Algorithm 9, on receiving the *findReply* tuple from Denise, Charlie updates the tuple with $\mathbf{txid}'$ and forwards it back towards Alice on $seg_{AC}$. When Denise receives the *findReceive* tuple from Bob, she retrieves the encrypted challenge nonce and the symmetric key. She composes a *findReply* tuple with encrypted information for Bob and forwards it to the node on $seg_{DB}$. This message is forwarded back towards Bob by each intermediate node. At the

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

**Input :** CI
**Output :** *findReply* tuples of *RH*s
**Parties :** Sender: Alice, Receiver: Bob

**for** $i \in [1..n]$ **do**
  /* Sender's *RH* sending reply */
  Charlie, on receipt of Denise's *findReply*() does:
  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.
  Compose reply to Alice:
  $findReply(\mathbf{txid}'||vk_C||E_{K_{AD}}(\mathbf{currMax}_s||y'||ts)||C_A)$.
  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 */
  In parallel, Denise, on receiving Bob's message will retrieve $(K_{BD}, y', \text{tS}) \leftarrow D_{DK_D}(C_B)$
  **if** *decryption fails* **then**
    do nothing.
  **end**
  **else**
    Reserve $\mathbf{currMax}_D$ by $\min(\mathbf{currMax}_r, \mathbf{currMax}_D)$, set $\mathbf{currMax}_r = \min(\mathbf{currMax}_r, \mathbf{currMax}_D)$.
    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 $\mathbf{txid}'''$.
  **end**
**end**

end of the ***Find Route*** phase, Alice and Bob both have a shared symmetric key with Denise, and can establish how much maximum credit can flow between Alice and Bob over $seg_{AC}$, $seg_{CD}$ and $seg_{DB}$.