SplitPayLock: MultiPath Payment Protocol in Payment Channel Network

Subhra Mazumdar‡, Sushmita Ruj∗

‡ Indian Statistical Institute Kolkata, India

∗ Data61, CSIRO, Australia

March 1, 2020

Abstract

Conventionally payment protocols developed to realize off-chain transactions in *Payment channel network (PCN)* assumes the underlying routing algorithm transfers the payment via a single path. However, a path may not have sufficient capacity to route a transaction. In such cases, it is inevitable to split the payment across multiple paths. Additionally, a protocol must ensure 'atomic payment' - *it either succeeds or fails in entirety.* In this paper, we propose a *secure* and *privacy-preserving* multipath payment protocol, *SplitPayLock*, ensuring *strong atomicity*, if all parties involved in routing are honest, and *weak atomicity*, if at least one intermediate party acts dishonestly. It is devoid of *zero-knowledge proof* and *complex key management.* A formal definition of the protocol in the *Universal Composability* framework has been discussed. We compare the performance of our protocol with state-of-the-art *Multihop HTLC* [1], applied iteratively on one path at a time on Ripple network and Lightning Network. It is observed that *SplitPayLock* requires less communication overhead and low execution time, demonstrating efficiency and scalability.

1 Introduction

Cryptocurrencies like Bitcoin [2] popular as an alternative method of payment. Blockchain forms the backbone of such a decentralized network, which not only allows transacting parties to remain pseudonymous but also guarantees reliability and security.. The records stored in this distributed ledger are immutable and can be verified by anyone in the network. It is replicated across users who use consensus algorithms like Proof-of-Work [2], [3], [4], Proof-of-Stake [5], [6]) for agreement on state change in the ledger. But such consensus algorithms are quite a computation and resource-intensive, slowing down the performance and reducing scalability [7], [8]. Thus scaling blockchain transactions is an important concern that needs to be addressed, without compromising with privacy.

Though several solutions [9], [10], [11] have been suggested, *Payment Channel* [10] stood out as a practically deployable answer to the scalability issue. Any two users, with mutual consent, can open a payment channel by locking their funds. Users can perform several off-chain payments between each other without recording the same on blockchain. This is done by locally agreeing on the new deposit balance, enforced cryptographically by smart contracts [8], key-based locking [12]. If one party wants

to close the payment channel, it broadcasts the transaction on blockchain with the final balance. If a counterparty tries to cheat by claiming payment for an old transaction, it will be penalized and lose out all the funds it had locked initially. Opening a new payment channel between parties that are not connected directly has its overhead where funds get locked for a substantial amount of time. This can be avoided by leveraging on the set of existing payment channels for executing a transaction, proving beneficial in terms of resource utilization. This set of payment channels form the *Payment Channel Network* or PCN [8].

Challenges faced. The major challenge in designing any protocol for PCN is to ensure the privacy of the payer and payee and hiding the payment value transferred. No party, other than the payer and payee, should get any information about the transaction. Thus any routing or payment algorithm designed for such a network must be decentralized, where individual nodes take decisions based on the information received from its neighbor. Finding a single route for routing the high-valued transaction did not serve the purpose when the capacity of all payment channels are very low. In such cases, it is better to split the high-valued transaction and transmit it over different paths. It eliminates the constraint of finding out a single route from sender to receiver with sufficient channel capacity to support larger payment. Several distributed routing algorithms [13], [14], [15], [16], [17], [18], [19] have been proposed. It was first mentioned in Elias et al. [13] that push relabel fits better as a routing algorithm for PCN because it proceeds locally, taking into account the residual capacity of each payment channel. Such a routing algorithm tend to split the transaction value across multiple paths, making the best possible utilization of each path.

The payment algorithms stated so far in [20], [21], [22], [1], [12] deal with the transfer of payment across a single path. Simply extending the existing payment protocol for multipath payment results in leakage of sensitive information, thereby allowing dishonest intermediaries to steal funds. The routing algorithm returns paths that may share several payment channels. Funds in these payment channels are locked repeatedly for a substantial amount of time upon implementing the payment protocol on each path, as shown in Section 4.2. This drastically reduces the performance as other payment requests are denied from using such payment channels. To avoid this problem, we initially map multiple paths involved in transferring the fund into a set of payment channels with a definite ordering, discussed later in Section 4.2. We propose an efficient privacy-preserving payment protocol on this set so that funds in each channel is locked not more than once. Our protocol allows all time-locked contract on incoming payment channels of a particular node to be released provided all the time-locked contract on its outgoing payment channel gets released first. We claim that our protocol ensures *Strong Atomic Multipath Payment* if all the parties act honestly. A payment is considered to be *strongly atomic* if the balance of all payment channels in a path is updated or no payment channel is modified [23]. However, even if one intermediate party acts dishonestly, by providing a wrong solution or going offline intentionally, during the period, we can guarantee *Weak Atomicity* of payment, as defined in [24], [25]. In that case, only the receiver claims the payment once it gets a guarantee of all the partial payment flow across all the paths involved in routing.

1.1 Goals of the Protocol

- **Transaction Privacy** - None of the intermediate nodes involved in routing a payment must be able to identify the sender and receiver. Since the payment value is split across multiple paths, we claim that intermediate nodes cannot figure out the total amount of the transaction.

- **Consistency** - The protocol is consistent if none of the nodes can claim funds from the predecessor contract without obtaining the solution from the successor time-locked contracts. Non-

adjacent parties, upon collusion, cannot unlock their contracts by bypassing honest intermediaries.

- **Atomicity** - We define two notions of atomicity under different adversarial models.
  - *Strong Atomicity* - A payment is said to be strongly atomic if it either succeeds or fails in entirety. This implies that in case of successful payment all the channels involved in routing must be updated or it must roll back to its previous state upon failure. We prove that our protocol achieves *Strong Atomicity*, if all the parties act honestly, under Claim 1 in Section 5.5.
  - *Weak Atomicity* - A payment is said to be weakly atomic if the receiver can claim the payment upon receiving all the partial payment flow. The payment channels involved in routing aggregate their individual secrets and provide it to the receiver. Upon receiving this value, the receiver can withdraw funds from the network. If any of the party misbehaves and does not lock fund then the transaction fails. However, once the receiver has claimed the payment, our protocol definition does not bother whether the rest of the parties involved in routing can claim their payment if an intermediate node behaves maliciously.
- **Non-interactive Setup** - Apart from sharing the information on the transaction value with the receiver, the sender need not coordinate with the receiver for setting up a protocol instance.

1.2 Our Contributions

- We have proposed a privacy-preserving off-chain payment protocol, *SplitPayLock*, for secure transactions across multiple paths in the payment channel network. The setup phase of the payment protocol is devoid of the complex key management process and zero-knowledge proof.
- Our protocol achieves *Strong Atomicity* if all nodes involved in the routing act honestly and *Weak Atomicity* if at least one intermediate node deviates from the protocol.
- We have defined the privacy notions of *SplitPayLock* based on the Universal Composability framework and provided a detailed security analysis. The security of our proposed scheme depends on the discrete logarithm problem in random oracle.
- We have implemented the proposed protocol on real instances - Ripple Network [1] and Lightning Network [26]. The code is given in [27]. *SplitPayLock* takes around 10s to complete the payment with a communication overhead of less than 1.5 MB compared to Multihop HTLC, which takes around 65s to complete the protocol and incurs a communication overhead of 26 MB. In an instance of Lightning Network, it takes around 485ms and communication overhead of 0.16 MB as compared to 10.6s and communication overhead of 24 MB by Multihop HTLC.
- The proposed payment protocol is modular and functionally independent and hence works perfectly for any underlying routing algorithm.

1.3 Organization

Section 2 discusses the state-of-the-art in PCN. Section 3 gives a brief overview of the preliminaries. Section 4 defines the problem statement and motivation. Section 5 gives a detailed overview of the construction of *SplitPayLock* and a formal definition of the security under Universal Composability

Model is given in Section 5.4. A generic construction of *SplitPayLock* is stated in Section 5.5 and Privacy Analysis is provided in 5.6. Performance Analysis of *SplitPayLock* protocol is provided in Section 6 and Section 7 concludes the paper.

2 Related Work

Payment Channel Network is a peer-to-peer, path-based transaction (PBT) network where each party operates independently of other parties. Several P2P path-based transaction networks such as such as Lightning Network for Bitcoin [8], Raiden Network for Ethereum [28], SilentWhispers [15], InterLedger [29], Atomic-swap [30], TeeChain [31] etc. have been developed over the years. Perun [32] proposes a more efficient network structure that is built around payment hubs. An extension of such networks, State Channel Network [33], not only supports off-chain payment but allows execution of complex smart contract. Spider network [34] adheres to a packet-switched architecture for payment channel networks.

Previously routing algorithm focussed on finding a single path for routing a transaction and were centralized in nature. Canal [17] uses a centralized server for computing the path, Flare [14] requires intermediate nodes to inform the source node about their residual capacity. However, in order to preserve the transaction privacy, it was not in the best interest to have a single coordinator with all information control the routing algorithm. Apart from this, payment value to be transferred can be larger than the capacity of any payment channel present in the network. In such cases, it is better to split the high-valued transaction and transmit it over different paths. Several routing algorithms like SilentWhispers [15] - a distributed PBT network without using any public ledger guaranteeing unlinkability of the transaction, SpeedyMurmur [16] - a privacy-preserving embedded based routing, extending Voute [35], depending on the presence of landmark nodes, addressed this issue. Apart from this, Elias et al. [13] proposed an extended push relabel for finding payment flow in the payment network, which took into account the capacity of each channel for deciding the number of splits. Later, a distributed approach for PCN routing, CoinExpress [18], was proposed for finding routes that fulfill payment with a higher success ratio. A routing algorithm based on swarm intelligence, ant colony optimization [36] has been explored. Hoenisch et al. [19] proposed an adaptation of an Ad-hoc On-demand Distance Vector (AODV)-based routing algorithm which supports different cryptocurrencies allowing transactions across multiple blockchains.

Privacy guarantee offered by PCN and its challenges has been extensively discussed in [37], [38], [39]. A payment along a path must be atomic - either it succeeds fully or it is aborted. Partial satisfaction of a transaction may lead to loss of funds. As a solution, Hashed Time-lock Contract [8] was proposed for Lightning Networks. It is compatible with the Bitcoin script but has its demerits. Bolt [21] states about a hub-based payment construction retaining payment anonymity but it is restricted to just two-hop payment. TumbleBit [40] follows a similar approach assuring payer/payee privacy but suffers from the same shortcoming. Sprite [22], a variant of payment channel network, does not focus on privacy and concurrency issues.

Malavolta et al. [1] had proposed a secure version of payment for multi-hop path based on zero-knowledge proof system `ZK-Boo` [41]. It uses HTLC as the backbone, working on one path at a time. Zero-knowledge proofs are used to prove that the preimage for outgoing HTLC of a node will assist in constructing the preimage for the incoming HTLC. *Multi-hop Locks*, defined in [12], are compatible with vast majority of cryptocurrencies. It is generic as well as interoperable, supporting both script and scriptless support for PCN. An efficient privacy-preserving payment protocol based on Chameleon Hash Function [42] was proposed which is devoid of complex key management and zero-knowledge

proof. But in this protocol, honest intermediaries lying on a path are susceptible to key exposure attacks. However, all such payment protocols deal with routing transactions via single path.

State-of-the-art on atomic multipath payment by Osuntokun [24] lacks security model and susceptible to *Wormhole attack* [12], as discussed in Section 4.1. Another payment protocol used on top of multipath routing, Boomerang [25] mentions that if payment is split across k paths and sender uses $k+n, n>0$ paths for the transaction then the receiver must cancel the microtransactions on n redundant paths if it receives confirmation from k paths. However such n paths are susceptible to *Griefing attack* [23], [43] as intermediate nodes may withhold the cancel message from being propagated to the sender. Both the protocols achieve *strong atomicity* in the presence of all honest parties and *weak atomicity* if an intermediate node misbehaves, but still the receiver can withdraw the full payment.

Several other protocols for cross-chain payment [44] and atomic multi-channel update have been studied in [23] but there is substantial leakage of information violating transaction privacy. The protocol Atomic Multi-Channel Update (AMCU) as defined in [23] guarantees strong atomicity across multiple paths, irrespective of intermediate parties acting honest or malicious. However, this payment protocol violates relationship anonymity and expects all the parties involved in routing to collaborate. Recently, a new technique based on *Dynamic Internal Payment Splitting* [45] recursively splits payments across multiple intermediaries in state channel network and receiver aggregates such payment receipts for claiming payment. But this protocol does not strictly adhere to the requirement of atomic transfer of payment and it will not work for the Bitcoin-based payment channel network.

3 Background

In this section, we provide the required background on the payment channel network. The terms source/payer means the sender node. Similarly, sink/payee/destination means the receiver node and transaction mean payment transfer.

3.1 Payment Channel Network

Definition 1 *A Payment Channel Network (PCN) [1] is defined as a bidirected graph $G := (V, E)$, where V is the set of accounts dealing with cryptocurrency and E is the set of payment channels opened between a pair of accounts. A PCN is defined with respect to a blockchain. Apart from the opening and closing of the payment channel, none of the transaction gets recorded on the blockchain. Upon closing the channel, cryptocurrency gets deposited into each user's wallet according to the most recent balance in the payment channel.*

Basic operations of PCN [1]-

- `openPaymentChannel`(v_1, v_2, α, t, m) : For a given pair of accounts $v_1, v_2 \in V$, channel capacity α (initial balance escrowed), timeout value of t and processing fee charged m, `openPaymentChannel` creates a new payment channel $(id_{(v_1,v_2)}, \alpha, t, m) \in E$, where $id_{(v_1,v_2)}$ is the channel identifier, provided both v_1 and v_2 has authorized to do so and the funds contributed by each of them sum up to value α.

- `closePaymentChannel`$(id_{(v_1,v_2)}, \tilde{\alpha})$: Given a channel identifier $id_{(v_1,v_2)}$ with balance $\tilde{\alpha}$, `closePaymentChannel` removes the channel from G provided it is authorized to do so by both $v_1, v_2 \in V$. The balance $\tilde{\alpha}$ gets written on blockchain and this amount is distributed between v_1 and v_2 as per the net balance recorded.

- `payVal`$(p(s,r), val)$: $p(s,r)$ denotes a path between sender s and receiver r. It is defined by a set of identifiers $id_{(s,v_1)}, id_{(v_1,v_2)}, \ldots, id_{(v_n,r)}$, where $s, v_1, v_2, \ldots, v_n, r \in V$, having enough credit to allow transfer of val from s to r, if for each payment channel denoted by $id_{(v_i,v_{i+1})}$ has capacity of at least $\beta \geq val'_i, val'_i = val + \Sigma_{j=i+1}^{n} fee(v_j), 0 \leq i \leq n, v_0 = s \; and \; v_{n+1} = r$, where $fee(v_j)$ is the processing fee charged by each intermediate node v_j in $p(s,r)$.

3.2 Hashed Time-lock Contract

Hashed Time-lock Contract (HTLC) [8], [1] is a payment protocol which allows secure transfers of funds from payer S to payee R, using a network of channels across an n-hop route $(u_0, u_1, u_2, \ldots, u_n), S = u_0, R = u_n$. The payee R creates a condition y defined by $y = \mathcal{H}(\tilde{x})$ where $\tilde{x}$ is a random string and $\mathcal{H}$ is a random oracle [46]. The condition y is shared with S. It now shares the condition with $u_1, u_2, \ldots, u_n$ for construction of the HTLC. Between any pair of adjacent nodes (u_i, u_{i+1}), the hashed time-lock contract is defined by $HTLC(u_i, u_{i+1}, y, b, t)$ - implies that this contract locks b units of fund of party u_i that can be released to party u_{i+1} only if it releases the correct preimage $\tilde{x}$ for which $y = \mathcal{H}(\tilde{x})$ within timeout t. This HTLC is then set at each payment channel present on the path to the receiver.

3.3 Multi-Hop Hashed Time-lock Contract

In HTLC construction (refer Section 3.2), the anonymity of the sender and receiver is not retained if we use the same hash value $\mathcal{H}(R)$ across all the contracts for release of payment. Plus, if intermediate nodes behave maliciously and upload the HTLC statements on the blockchain, any observer can easily extract the complete routing information. Thus, as a solution, *Multi-Hop HTLC* was proposed to address privacy concerns.

Given n-hop route $(u_0, u_1, u_2, \ldots, u_n)$ between a sender and receiver, sender first samples n independent strings $(x_1, x_2, \ldots, x_n)$. Then, the following values are constructed by the sender [1]:

$$\forall i \in 1, 2, \ldots, n; \; y_i = \mathcal{H}(\bigoplus_{j=i}^{n} x_j) \tag{1}$$

where H is a random oracle. The sender provides the receiver with (y_n, x_n) and the i^{th} node is supplied with the tuple $T_i = (y_{i+1}, y_i, x_i), \forall 1 \leq i < n$. In order to show that T_i is well formed, statement to be proved is as follows '$\exists x : y_{i+1} = \mathcal{H}(x) \wedge y_i = \mathcal{H}(x_i \oplus x)'$. Hence a zero-knowledge proof Π_i [41] of the form

$$\Pi_i \leftarrow \mathcal{P}\Big((\mathcal{H}, y_{i+1}, y_i, x_i), \Big(\bigoplus_{k=i+1}^{n} x_k\Big)\Big) \tag{2}$$

is send to node i along with the tuple T_i. Payment protocol is aborted if the proof is not formed correctly. Contract $(u_i, u_{i+1}, y_{i+1}, v, t_i)$ is setup on the channel between users u_i and u_{i+1} where time $t_i = t_{i-1} - \Delta$, t_{i-1} is the timeout parameter for the preceeding payment channel (u_{i-1}, u_i), for some fixed $\Delta > 0$.

3.4 Discrete Logarithm Problem

Cosnider an elliptic curve $\mathbb{G}$ over a finite field $\mathbb{F}_q$, where $q = p^n$ and p is prime. The elliptic curve discrete logarithm problem (ECDLP) is the following computational problem: Given points $P, Q \in$

$\mathbb{G}(\mathbb{F}_q)$, find an integer a such that $Q = aP$, if a exists. This computational problem is called the *Elliptic Curve Discrete Logarithm Problem* which forms the the fundamental building block for elliptic curve cryptography [47].

4 Problem Statement & Motivation

A high-valued transaction is split across multiple paths. Considering this mode of routing, we define the problem as follows -

Problem 1 *Given a payment channel network $G(V, E)$, a transaction request (s, r, val) for a source-sink pair (s, r) uses set of paths $p_1, p_2, \ldots, p_m$ in order to transfer the fund such that p_1 transfers val_1, p_2 transfers $val_2, \ldots$, p_m transfers $val_m : val = \Sigma_{i=1}^{m} val_i$. The objective is to design an efficient payment protocol ensuring atomic payment across all the paths without violating transaction privacy i.e. neither the sender nor the receiver of a particular transaction must be identified as well as hiding the actual transaction value from intermediate parties.*

Conjecture 1 *A payment protocol cannot achieve both strong atomicity as well as transaction privacy.*

The conjecture was made after studying the state-of-the -art. In [22], the feature of constant locktimes across all the payment channel in a path relied on a global contract mechanism. Since smart contracts used in Ethereum have more flexibility compared to Bitcoin based scripts, it was conjectured that Bitcoin based PCN cannot achieve the objective constant locktime without some modification to its scripting system. Later Egger et al. [23] came up with a construction for atomic multi-channel update with constant collateral without modifying the scripting system in Bitcoin. All participants are aware of the channels used in the protocol due to the use of the MIMO transaction. Both the schemes guarantee strong atomicity but at the cost of using public bulletin or global event, revealing the identity of the participants. Additionally, to guarantee consistency of state across all the payment channels, the parties involved in routing must cooperate. Non-cooperation eventually leads to failure of payment. This violates transaction privacy. On the other hand, the payment protocol as stated in [24], [25] claim to be 'atomic' but their definition of atomicity is not as rigorous as the one defined in [23]. In the next section, we discuss possible attacks on these two constructions.

4.1 Problems encountered in existing Atomic Multipath Payment protocol

As per the construction of Atomic Multipath Payment [24], we explain the steps via an example as shown in Fig.1. Given there exist two paths $P_1 = \langle S, A, M_1, \ldots, B, M_2, R\rangle$ and $P_2 = \langle S, P, Q, X, R\rangle$ from S to R, hashed time-lock contract is used on these two paths using the value H_1 and H_2 where $H_1 = \mathcal{H}(p_1)$ and $H_2 = \mathcal{H}(p_2)$. Here $p_1 = \mathcal{H}(BP||1)$ and $p_1 = \mathcal{H}(BP||2)$ where BP is the base preimage. S includes value s_1 and s_2 in the extra onion blobs (EOBs) to be opened by the receiver. Receiver reconstructs $BP = s_1 \oplus s_2$ and uses it to reconstruct preimages r_1 and r_2 for paths P_1 and P_2 respectively. For the path P_1, R shares r_1 with M_2. Since M_1 and M_2 are controlled by the same adversary, instead of forwarding r_1 to B, it forwards the secret directly to M_2. All the intermediate parties from A to B think that the payment has failed and withdraw their money after the timeout period. On the other hand, M_1 forwards the secret to S. Both the parties M_1 and M_2 gain the processing fee of the intermediate nodes. This is known as the *Wormhole Attack*, defined in [12].

The same problem is faced in Boomerang by Bagaria et al. [25]. Consider the payment as shown in Fig. 2 where a quorum of two out of four paths is needed for the transfer of funds. The four phases -

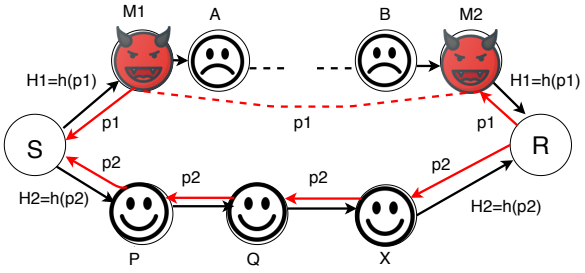


Figure 1: Wormhole Attack in path P_1

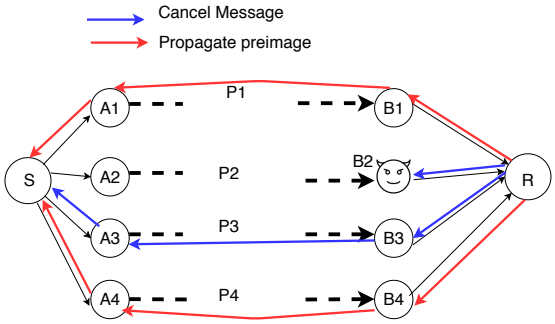


Figure 2: Griefing Attack in path P_2

Promise, Deliver, Cancel and *Finish* are carried out in the subgraph. Two extra redundant paths are added. In the *Promise* phase, hash time-locked contract is established for each path P_i as $H_i = \mathcal{H}(p_i)$. Each p_i is the secret share of a polynomial (using verifiable secret sharing). In the *Deliver* phase, P_1 and P_4 are used for transfer of funds. The receiver reveals the preimage for these two paths. B requests cancellation of two paths P_2 and P_3. It might be the case that node B_2 in P_2 is malicious and doesn't forward the message for canceling the HTLC to other nodes. Then the resources in path P_2 remain locked inducing a high collateral cost.

4.2 Mapping set of routes into set of Payment Channels

To mitigate the problems stated above, it was explored whether existing privacy-preserving protocols like [1], [12] can be applied sequentially over multiple paths, considering one path at a time. However, the following problems were encountered -

- Given m paths, starting from path p_1, the algorithm is applied iteratively on each path $p_i, 1 \leq i \leq m$. Once the payment protocol succeeds in path p_i, the capacity of the path is reduced by val_i. In the worst case, if payment fails in the m^{th} path, then just $\Sigma_{i=1}^{m-1} val_i$ gets transferred to the receiver r. This does not guarantee even weak atomicity. On the other hand, even if a payment protocol succeeds in all the m paths, the procedure needs to be repeated for each m route thereby increasing computation overhead by a factor of m. It is not a good approach for non-edge-disjoint paths were funds of the shared payment channel gets blocked for a prolonged period.

Consider the payment channel network given in Fig. 3 for transferring funds from A to F. The routing algorithm returns two paths given as $p_1 = A \to B \to C \to E \to F$ and $p_2 = A \to B \to D \to E \to F$. If we apply payment protocol first to path p_1 and then to path p_2, funds in channels AB and EF are locked twice.

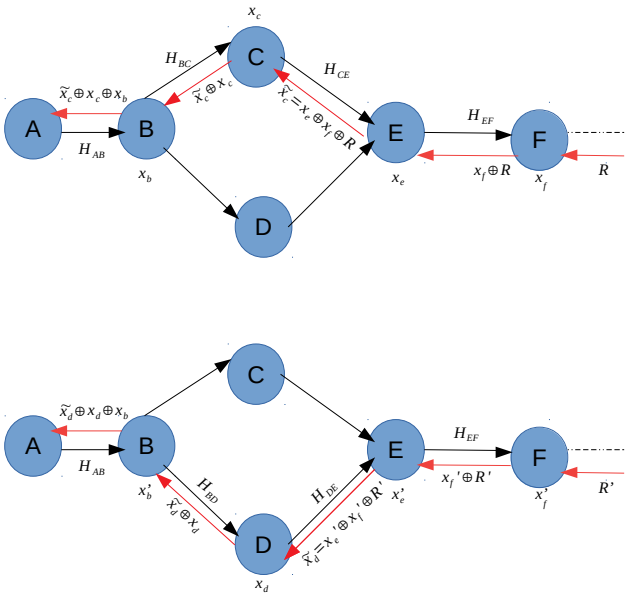


Figure 3: Multihop HTLC applied first on route p_1 and then on p_2

To alleviate the problem mentioned above, we map the set of m paths into a set of payment channels. Given the set of routes, denoted by $\mathcal{P}$, the process of ordering payment channels is stated in Procedure 1. Following the procedure, we map the two paths p_1 and p_2 in Fig. 3 into set of payment channels, starting from receiver node F. It has one incoming contract established on payment channel EF. This channel is inserted into the set $\mathbb{PC}$. F is inserted into the queue Q. This continues till $\mathcal{P}$ is empty. The set constructed is $\mathbb{PC} = \{EF, DE, CE, BD, BC, AB\}$.

We propose a new payment protocol for the set $\mathbb{PC}$, *SplitPayLock*, which ensures transaction privacy. Under the setting of all parties acting honestly, the protocol achieves *strong atomicity* across all the paths. Under a malicious setting, the protocol achieves *weak atomicity*, where it is deemed sufficient for the receiver to successfully claim the fund. To ensure consistency, a node must release lock for all its incoming contracts if and only if it receives solutions from the outgoing neighbors with whom it had formed a contract.

5 Our Proposed Construction

5.1 Network Model and its Assumptions

- The topology of the network is known by any node in the network since any opening or closing of a channel is recorded on the blockchain.

Procedure 1: Mapping set of paths $\mathcal{P}$ into set of payment channels

Input: $\mathcal{P}$
Output: $\mathbb{PC}$
Initialize set $\mathbb{PC} = \phi$ and a queue $Q \leftarrow \phi$.
Insert the receiver node r into Q.
while *Q is not empty* **do**
    $v \leftarrow Q.pop()$. Mark v as *visited*.
    Find out the incoming neighbours of v with which it had established time-locked contract in order to route the payment.
    Insert these payment channel, with v as the counterparty, into the set $\mathbb{PC}$ and delete it from the set $\mathcal{P}$.
    Insert all *unvisited* incoming neighbours of v into Q.
end

- For simplicity of analysis, we allow at most one time-locked contract to be established on a payment channel at a time.
- Sender of a payment chooses a set of paths to the receiver according to her own criteria.
- The current value on each payment channel is not published but instead kept locally by the users sharing a payment channel.
- Every user is aware of the payment fees charged by each other user in the PCN.
- Pairs of users sharing a payment channel communicate through secure and authenticated channels.
- An honest node in the network does not go offline during execution of a protocol instance.

5.2 Definition

Formally, *SplitPayLock* is defined with respect to a universe of users $\mathbb{U} = \mathbb{U} = \{U_i\}, i \in [0, n]$, U_0 denotes the initiator of protocol and U_n denotes the receiver. It is a 4-tuple of PPT algorithms defined as follows:

Definition 2 *SplitPayLock protocol* $\mathbb{SPL} = (KGen, Setup, TimeLockContractCreate, TimeLockContractRelease)$ *consists of the following efficient algorithms-*

- $(sk_{U_i}, pk_{U_i}) \leftarrow KGen_{U_i}(1^\lambda)$
  Input : Security parameter 1^λ*, Output : Public key* pk_{U_i} *and corresponding private key* sk_{U_i} *for user* U_i*.*
- $(m_1, m_2, \ldots, m_n) \leftarrow Setup_{U_0}(1^\lambda, U_1, \ldots, U_n)$
  Input : Security parameter 1^λ*, vector of identities* $(U_1, U_2, \ldots, U_n)$*, Output : Protocol randomly samples* x_{U_i} *returns, for* $i \in [1, n]$*, creates message* m_{U_i} *for each user* U_i*, where* m_{U_i} *contains* $(\overrightarrow{R_{in,U_i}}, x_{U_i}, \overrightarrow{R_{out,U_i}})$*.* $\overrightarrow{R_{in,U_i}}$ *is the vector of commitment values based on which incoming neighbours of* U_i *created contract with* U_i *and* $\overrightarrow{R_{out,U_i}}$ *is the vector of commitment values based on which* U_i *created contract with outgoing neighbours of* U_i*.*

- $R_{U_i,U_j} \leftarrow TimeLockContractCreate(m_{U_i}, U_i, U_j)$
  Input : m_{U_i} *contains* $(\overrightarrow{R_{in,U_i}}, x_{U_i}, \overrightarrow{R_{out,U_i}})$, *identities* U_i *and* U_j, *Output :* U_i *checks whether there exists a relation between* $x_{U_i}, \overrightarrow{R_{in,U_i}}$ *and* $\overrightarrow{R_{out,U_i}}$. *If so, then it forms a hashed time-lock contract with outgoing neighbour* U_j *using commitment value* $R_{U_i,U_j} : R_{U_i,U_j} \in \overrightarrow{R_{out,U_i}}$.
- $r_{U_i,U_j} \leftarrow TimeLockContractRelease(R_{U_i,U_j}, U_i, U_j)$
  Input : The commitment value R_{U_i,U_j}, *identities* U_i *&* U_j, *Output : The decommitment value* r_{U_i,U_j}.

5.3 Key Idea

For the *Key Generation* phase, assume that each node in the network generates its public key-private key pair at the joining phase itself. As shown in Fig.4, the routes for a particular transaction from sender to receiver are first mapped into a set of payment channels $\langle SA, AB, BT \rangle$ and provided as input to the proposed payment protocol.

In the *Setup* Phase, the sender node provides the necessary information for forming the contract with its adjacent neighbour for claiming funds. In this phase, sender also provides additional information to each node which guarantees correctness of partially reconstructed key needed by the receiver in order to trigger the *TimeLockContractRelease* phase. In Fig.4, S provides (R_A, x_a, R_B, X_A) to A, (R_B, x_b, R_T, X_B) to B and (R_T, x_t, ϕ, X_T) to T.

In *TimeLockContractCreate* Phase, a node checks whether a relation holds between the commitment used for its incoming contract and its outgoing contract. For example, in Fig.4, node A checks that contract R_A and R_B must be related. This guarantees that each node can claim funds upon releasing the correct solution. Apart from this, a node has to share its secret value as well for partially reconstructing the key needed by the receiver. In Fig.4, S provides $f(x_s)$ to A, f is an operation used for combining partial secrets. Using $f(x_s)$, A computes $f(x_s, x_a)$ and shares it with B and this goes on till it reaches the receiver node T.

In order to trigger *TimeLockContractRelease* Phase, using the information from all the nodes involved in routing, receiver T constructs the key $k = f(x_s, x_a, x_b, x_t)$. If any of the parties misbehaved in the process then the receiver will not release the lock. Using k, T constructs r_t for claiming the payment. Using r_t, node B will construct r_b and release it to A. This guarantees atomic payment whereby all intermediate nodes are forced to lock their money and participate in the construction of the key.

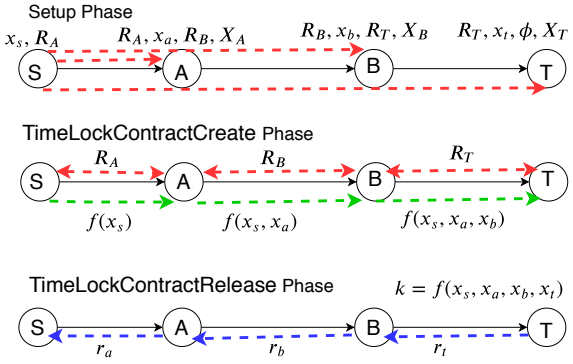


Figure 4: SplitPayLock Protocol

5.4 Security Definitions

5.4.1 Assumptions made on Security Model

For modeling security and privacy definition of payment across several payment channels under concurrent execution of an instance of *SplitPayLock*, we take the help of Universal Composability framework, first proposed by Canetti et al. [48].

5.4.2 Attacker Model

Using the model suggested in [1], the nodes of the network are modeled as interactive Turing machines, denoted by $\mathbb{U} = \{U_i\}, i \in \tilde{V}$, U_0 denotes the initiator of protocol and U_n denotes the receiver, which communicates with an ideal functionality $\mathcal{F}$ via secure and authenticated channels. We model the attacker $\mathcal{A}$ as a `PPT` machine that is allowed to corrupt a subset of nodes in the network. Upon corruption, it gets access to its internal state and controls any transmission of information to and from the corrupted node. As of now, only static corruption is allowed, i.e. adversary must specify the nodes it wants to corrupt before the start of the protocol.
Adversarial Goal - An adversary tries to stall the network by launching *Denial of Service (DoS)* attack. In such cases, an adversary may forgo any incentive it would have received by following the protocol since it is strongly motivated to disrupt the network [43].

5.4.3 Communication Model

For encoding the secure communication and anonymous communication between two parties in the ideal world, we define it in the following way -

- Using secure message transmission functionality $\mathcal{F}_{smt}$, U_i sends packet $(sid, \text{instruction}, U_i, U_j, m)$, containing the secret message m to U_j. $(sid, \text{instruction}, U_i, U_j, |m|)$ is leaked to Sim, where $|m|$ is the length of the message, without revealing the content of the message [48].
- Using anonymous message transmission functionality $\mathcal{F}_{anon}$, U_i sends packet $(sid, \text{instruction}, U_i, U_j, m)$, containing the secret message m to U_j. $(sid, \text{instruction}, U_j, |m|)$ is leaked to Sim [48], [49], without revealing the content of the message and the identity of the sender.

In both cases, an attacker can delay the delivery of messages arbitrarily. The network model is assumed to be synchronous [48], [50], where any message sent out at i^{th} round, gets delivered to the intended recipient at $(i+1)^{th}$ round. This is modeled using the functionality $\mathcal{F}_{syn}$. Computation in this model is assumed to be instantaneous. However, since we deal with the asynchronous network in the real world, a maximum time bound for message transmission is set. If no message is delivered by the predecided expiration time, then the message is considered as $\perp$.

Definition 3 ***UC Definition of Security****. An environment $\mathcal{Z}$ present in both the ideal and real world invokes the steps of execution of an instance by providing the input and receiving the output. If $\mathcal{Z}$ cannot distinguish between Π in the real world and the ideal world functionality, then it is said to be UC-secure. Formally stated,*

Theorem 1 *Given that λ is the security parameter, a protocol denoted by Π, UC-realizes an ideal functionality $\mathcal{F}$ if for all computationally bounded adversary $\mathcal{A}$ attacking Π there exist a probabilistic polynomial-time simulator Sim such that for all probabilistic polynomial time environment $\mathcal{Z}$ such that $IDEAL_{\mathcal{F},Sim,\mathcal{Z}}$ and $REAL_{\Pi,\mathcal{A},\mathcal{Z}}$ are computationally indistinguishable.*

5.4.4 Ideal World Functionality

We define an ideal functionality $\mathcal{F}$ for *SplitPayLock*. A list denoted by $\mathcal{C}$ is used to to maintain the set of time-locked contracts formed. The entries in $\mathcal{C}$ are of the form $(\overrightarrow{U_{in,U_i}}, \overrightarrow{con_{in,U_i}}, U_i, init, \overrightarrow{U_{out,U_i}}, \overrightarrow{con_{out,U_i}})$ where $\overrightarrow{U_{in,U_i}}$ is the vector of incoming neighbours of U_i, U_i is one of the parties, $\overrightarrow{U_{out,U_i}}$ is the vector of outgoing neighbours of U_i, $\overrightarrow{con_{in,U_i}}$ is a vector of incoming unique contract identifiers to be established with U_i and $\overrightarrow{con_{out,U_i}}$ is a vector of outgoing unique contract identifiers formed by U_i with $U_k : U_k \in \overrightarrow{con_{out,U_i}}$. Other entries in $\mathcal{C}$ are of the form $(U_i, U_j, status, con_{U_i,U_j})$ where $status \in \{Create, Release\}$, con_{U_i,U_j} is the unique contract identifier for the contract established on the payment channel between U_i and U_j. For extracting information on contract established given the identifier, such as the contract's status (*fetchContractStatus()*), the counterparties of a payment channel on which the contract was established (*fetchLeftNode(), fetchRightNode()*) and list of all contract identifier of all outgoing contracts formed by a node (*fetchOutgoingContract()*). In order to update the status of the contract, *updateContractStatus()* is used.

The interface of $\mathcal{F}$ is defined in Fig. 5. We provide a high-level overview of the functions defined in the interface. *KeyGen* allows a user to generate a public key-private key pair. *Setup* allows user U_0 to communicate the message needed to form contracts to guarantee successful payment from U_0 to U_n, to every user $U_i \in \tilde{V}$. *FormContract* allows user U_i to form a contract with its outgoing neighbour in $\tilde{V}$. *Release* allows user U_i to release the solution, as specified by the terms of the contract, to its left neighbor. The contract identifier assigned to each contract is unique. For exchange of message through $\mathcal{F}_{smt}$, the interface $send_s$ and $receive_s$ is used and to send message through $\mathcal{F}_{anon}$, the interface $send_{an}$ is used.

Assuming all parties have access to $\mathcal{F}$ via the interfaces as defined in Fig. 5. The protocol SplitPayLock with underlying blockchain B provides a fully secure payment channel network. An ideal functionality $\mathcal{F}_B$ maintains B locally. B is updated as per the transaction between parties. Any user can send a *read* instruction to $\mathcal{F}_B$, where the whole transcript of B is sent as a reply. The number of entries of B is denoted by $|B|$. An arbitrary condition can be specified in the contract in order to execute a transactions in B. $\mathcal{F}_B$ is entrusted to enforce that a contract is fulfilled before the corresponding transaction is executed. Time is modeled as the number of entries of the blockchain B. By adding dummy entries to B, time can be elapsed artificially. Users figure out the current time by counting the entries of B.

5.4.5 Basic Operations of Payment Channel Network in Ideal World

Assuming atmost one payment channel to exist between a pair of users, we describe the basic operations in PCN - *open channel, close channel* and *payment*.

- OPEN CHANNEL : Considering a user U_i wants to opena a channel with U_j. U_i invokes $\mathcal{F}$ by sending the message $(sid, open, U_i,$
  $U_j, \alpha, t, fee)$, where α is the amount deposited in the channel, t is the timeout period of channel and fee is fee charged on using the channel. If the operation is succesful, then $(sid, opened, id_{(U_i,U_j)}, U_i,$ $\alpha, t, fee)$ is returned to U_j. When U_j sends an acknowledgement of the event to $\mathcal{F}$, it sends and event identifier (h, id_{U_i,U_j}) for the channel creation to both U_i and U_j. An initial blockchain deposit recorded in B, which includes the capacity, channel timeout and fee charged for using the channel. If the operation fails, $(sid, \perp, U_i, U_j, t, fee)$ is send to U_i.
- CLOSE CHANNEL : For a channel between U_i and U_j, if either of the party wants to close the channel, it invokes $\mathcal{F}$ with the message $(sid, close, id_{(U_i,U_j)}, h)$. If the event is succesful, U_j is

KeyGen(sid, λ)
Upon invocation by U_i:

- $sk_{U_i} \leftarrow \lambda$
- $pk_{U_i} = sk_{U_i}\mathcal{G}$
- $send_s(sid, U_i, pk_{U_i}, sk_{U_i})$ to U_i

FormContract(sid,*con*)
Upon invocation by U_i :

- If $fetchContractStatus(con) \neq init$ *or* $fetchLeftNode(con) \neq U_i$ then abort.
- $send_s(sid, con, Create)$ to $fetchRightNode(con)$
- $receive_s(sid, b)$ from $fetchRightNode(con)$
- if $b = \perp$, send $\perp$ to U_i and abort.
- $updateContractStatus(con, Create)$, $send_s(sid, con, Create)$ to U_i

Setup(sid, $\mathbb{U}, \mathbb{PC}$)
Upon invocation by U_0 :
$\forall U_i \in \tilde{V} \setminus \{U_0, U_n\}$:

- $\forall j \in \tilde{V}, (U_j, U_i) \in \mathbb{PC}, con_{U_j,U_i} \leftarrow \{0,1\}^*$
- $\forall k \in \tilde{V}, (U_i, U_k) \in \mathbb{PC}, con_{U_i,U_k} \leftarrow \{0,1\}^*$
- $\overrightarrow{con_{in,U_i}} = \{(U_j, U_i) \in \mathbb{PC}, con_{U_j,U_i}\}$
- $\overrightarrow{con_{out,U_i}} = \{(U_i, U_k) \in \mathbb{PC}, con_{U_i,U_k}\}$
- insert $(\overrightarrow{U_{in,U_i}}, \overrightarrow{con_{in,U_i}}, U_i, init, \overrightarrow{U_{out,U_i}}, \overrightarrow{con_{out,U_i}})$ in $\mathcal{C}$
- $send_{an}(sid, \overrightarrow{con_{in,U_i}}, \overrightarrow{con_{out,U_i}}, U_i, \overrightarrow{U_{in,U_i}}, \overrightarrow{U_{out,U_i}}, init)$ to U_i

insert $(\perp, \perp, U_0, init, \overrightarrow{U_{out,U_0}}, \overrightarrow{con_{out,U_0}}), (\overrightarrow{U_{in,U_n}}, \overrightarrow{con_{in,U_n}}, U_n, init, \perp, \perp)$ in $\mathcal{C}$.

$send_{an}(sid, \perp, \overrightarrow{con_{out,U_0}}, U_0, \perp, \overrightarrow{U_{out,U_0}}, init)$ to U_0,
$send_{an}(sid, \overrightarrow{con_{in,U_n}}, \perp, U_n, \overrightarrow{U_{in,U_n}}, \perp, init)$ to U_n

Release(sid, *con*)
Upon invocation by U_i :

- if $fetchRightNode(con) \neq U_i$ *or* $fetchContractStatus(con) \neq Create$ *or* $fetchContractStatus(fetchOutgoingContract(con)) \neq Release$ *and* $fetchOutgoingContract(con) \neq \perp$ then abort
- $updateContractStatus(con, Release)$, $send_s(sid, con, Release)$ to fecthLeftNode(con)

Figure 5: Interface of ideal world functionality $\mathcal{F}$

informed about it and the users update their bitcoin balances in B as per the final channel state. From this point on, U_i and U_j must ignore all the requests from $\mathcal{F}$ relative to (h, id_{U_i,U_j}).

Set of paths $\mathcal{P}$ is mapped into a set of open payment channels $\mathbb{PC}$. A payment operation transfers a value v from a sender, party U_0, to a receiver, party U_n via channels in $\mathbb{PC}$. The payment routine for Sender is defined in Procedure 2. For Procedure 3, each intermediate node checks whether the capacity of the channel is high enough to support the transfer of the coins and whether the timeouts give by the sender are consistent. When the terms of contract reach the receiver U_n, it checks whether the transacted value is what it expects and whether the latest timeout t_n is well-formed (Procedure 4). If both conditions hold, the receiver releases the contract by querying the ideal functionality. This

triggers a cascade of release calls in the payment channels existing in $\mathcal{P}$, thereby enabling the users involved to withdraw their payment. If for some reason one of the intermediate payment channels is not released, then all the previous contracts abort after the corresponding timeout period.

Procedure 2: Payment Protocol for sender

Input: $(sid, pay, U_0, U_1, U_2, \ldots, U_n, \mathbb{PC}, v)$
$v_1 = v + \Sigma_{i=1}^{n} fee(U_i)$
if $v_1 \leq \Sigma_{\forall U_j \in \tilde{V}:(U_0,U_j)\in\mathcal{P}}\ cap(U_0, U_j)$ **then**
  Query $\mathcal{F}$ on *Setup(sid,*$(U_0, U_1, \ldots, U_n)$
  $\mathcal{F}$ returns $sid, \perp, \overrightarrow{con_{out,U_0}}, U_0, \perp, \overrightarrow{U_{out,U_0}}, init)$
  for $U_j \in V : (U_0, U_j) \in \mathbb{PC}$ **do**
    $cap(U_0, U_j) = cap(U_0, U_j) - v_1$
  end
  $t_0 = t_{now} + \Delta.l$, l is the maximum level traversed during bfs on the set of paths $\mathcal{P}$.
  for $U_i \in [1, n-1]$ **do**
    $t_{U_i} = t_{prev,U_i} - \Delta$, t_{prev,U_i}, is the timeout period of edges present in preceeding level of U_i
    $v_{U_i} = v_{prev,U_i} - fee(U_i)$, v_{prev,U_i}, is the value propagated forward after deducting fee of predecessors vertices in the path
    Send $(sid, forward, \overrightarrow{U_{in,U_i}}, \overrightarrow{U_{out,U_i}}, v_{U_i}, t_{prev,U_i}, t_{U_i})$ to U_i, $\overrightarrow{U_{in,U_i}}$ denotes all predecessors of U_i in $\mathbb{PC}$ and $\overrightarrow{U_{out,U_i}}$ denotes all succesors of U_i in $\mathcal{PC}$.
  end
  Send $(sid, forward, \perp, \overrightarrow{U_{out,U_0}}, v_1, t_0, t_0 - \Delta)$ to U_0
  Send $(sid, forward, \overrightarrow{U_{in,U_n}}, \perp, v, t_{now} + \Delta, t_{now})$ to U_n
  for $con \in \overrightarrow{con_{out,U_0}}, U_j \in \overrightarrow{U_{out,U_0}}$ **do**
    query $\mathcal{F}$ on FormContract(con)
    if $\mathcal{F}$ *returns* $(sid, con, Create)$ **then**
      contract$(U_0, U_j, con, v_1, t_0 - \Delta)$
    end
    else
      Send $\perp$ to U_j
    end
  end
end

The ideal functionality $\mathcal{F}$ ensures the following security and privacy properties for payment protocol:

- *Consistency*: *SplitPayLock* is consistent if no attacker can satisfy the terms mentioned in the incoming time-locked contract without receiving a solution from the outgoing time-locked contracts. $\mathcal{F}$ models this property with the help of *Release* interface. No adversary can bypass any intermediate node to release a contract.
- *Relationship Anonymity*: It requires that each intermediate node does not learn any information about the set of users apart from its direct neighbors. $\mathcal{F}$ ensures this as during *Setup* phase, contract

Procedure 3: Payment Protocol for each intermediate node

Input : $(sid, decision, \overrightarrow{U_{in,U_i}}, \overrightarrow{U_{out,U_i}}, v, t_{U_i}, t_{prev,U_i})$
if *decision is forward* **then**
  $\mathcal{F}$ returns $(sid, \overrightarrow{con_{in,U_i}}, \overrightarrow{con_{out,U_i}}, U_i, \overrightarrow{U_{in,U_i}}, \overrightarrow{U_{out,U_i}}, init)$
  for $U_j \in \overrightarrow{U_{out,U_i}}$, $con \in \overrightarrow{con_{out,U_i}}$ **do**
    if $v \leq c(U_i, U_j)$ *and* $t_{U_i} = t_{prev,U_i} - \Delta$ **then**
      $cap(U_i, U_j) = cap(U_i, U_j) - v$
      Query $\mathcal{F}$ on FormContract(sid, con)
    end
    else
      for $U_j \in \overrightarrow{U_{in,U_i}}$ **do**
        Send $\perp$ to U_j
      end
    end
  end
end
else if *decision is* $\perp$ **then**
  for $U_j \in \overrightarrow{U_{out,U_i}}$, $con \in \overrightarrow{con_{out,U_i}}$ **do**
    $cap(U_i, U_j) = cap(U_i, U_j) + v$
  end
  for $U_j \in \overrightarrow{U_{in,U_i}}$ **do**
    Send $\perp$ to U_j
  end
end
else if *(decision=OK)* **then**
  for $con \in \overrightarrow{con_{out,U_i}} : fetchContractStatus(con) = Release)$ **do**
    query $\mathcal{F}$ on $Release(sid, con)$
  end
  for $U_j \in \overrightarrow{U_{in,U_i}}$ **do**
    Send OK to U_j
  end
end
else
  for $U_j \in \overrightarrow{U_{in,U_i}}$ **do**
    Send $\perp$ to U_j
  end
end

Procedure 4: Payment Protocol for receiver

Input: $(sid, release, U_n, \mathbb{PC}, v_{U_n}, t_{U_n}, v)$
$\mathcal{F}$ returns $(sid, \overrightarrow{con_{in,U_n}}, \perp, U_n, \overrightarrow{U_{in,U_n}}, init)$
for $U_j \in \overrightarrow{U_{in,U_n}}$ **do**
  if $(t_{U_n} > t_{now} + \Delta) \wedge (v_{U_n} = v) \wedge (fetchContractStatus(con) = Create)$ **then**
    Query $\mathcal{F}$ on Release(sid, con)
    send OK to U_j
  end
  else
    send $\perp$ to U_j
  end
end

identifiers are sampled at random and a user only learns the identifiers of the contract formed with its outgoing neighbors and identifier of the contracts established by its incoming neighbors.

5.5 Generic Construction

Given a set of payment channels denoted by $\mathbb{PC}$ for routing the transaction (s, r, val) from s to r in a bidirectional payment channel network $G(V, E)$, not necessarily edge disjoint, comprising set of nodes $\tilde{V} \subseteq V$ with $|\tilde{V}| = n + 1$. Sender s is denoted by U_0 and r denoted by U_n. Consider an elliptic curve group with generator $\mathcal{G}$, with $|\mathcal{G}| = q$ and λ be the security parameter. For our protoocol instance, the operation f is defined as addition. We discuss each phase in details.

5.5.1 KeyGen Phase

Each node $v \in V$ independently samples a pair of public key and private key $(pk_v, sk_v) : pk_v = sk_v\mathcal{G}$, where $sk_v \xleftarrow{\$} \{0,1\}^\lambda$ and pk_v is a point on the elliptic curve. The public key is a long term key and it is used repeatedly across different instance of the protocol, until and unless the secret key gets compromised.

5.5.2 Setup Phase

Consider a function $\mathcal{H} : \{0,1\}^* \to \{0,1\}^\lambda$ as random oracle. U_0 samples $n + 1$ independent strings $x_i : x_i \in \mathbb{Z}_q, 0 \leq i \leq n$. Since receiver node is the one with no outgoing flow, funds of payment channel denoted by $id_{u,r}, \forall u \in \tilde{V}, (u, r) \in \mathbb{PC}$, with receiver node as one of the counterparty is locked for the least time. Let this be t_0. For hashed time-lock contracts established with any other pair of nodes (v, u), check the value $t_u = t_{v,u} = max\{t_{u,w} : \forall w \in \tilde{V}, (u, w) \in \mathbb{PC}\} + \Delta$ for some positive value of Δ. Assign $t_{v,u}$ as the lock time for the contract on payment channel (v, u). For each channel $(u, w), u, w \in \tilde{V}$, denoted by $id_{u,w}$, a commitment for locking funds is constructed in the following way -

$$R_{u,w} = x_w\mathcal{G} + e_{u,w}.pk_w + \Sigma_{(w,i)\in\mathbb{PC}:i\in\tilde{V}}R_{w,i} \tag{3}$$

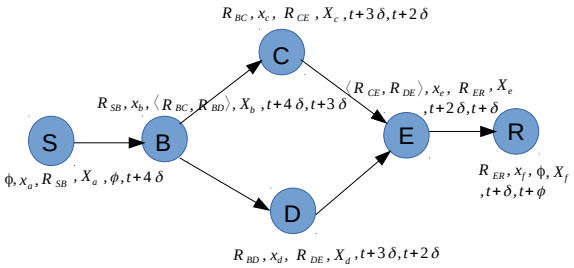


Figure 6: SplitPayLock : Setup

where pk_w is the public key of w. $e_{u,w}$ is constructed as

$$e_{u,w} = \mathcal{H}(x_w \mathcal{G} + \Sigma_{(w,i)\in\mathbb{PC}:i\in\tilde{V}} R_{w,i} || id_{u,w}) \tag{4}$$

If $w = U_n$ then $R_{w,i} = \phi$ and $x_w = \tilde{x_w}$, where $\tilde{x}_w$ is defined in Eqn. 6. For each node $U_i, 0 \leq i \leq n$, we construct X_i

$$X_i = (\Sigma_{(U_j,U_i)\in\mathbb{PC}:U_j\in\tilde{V}} \tilde{x_j} + x_i)\mathcal{G} \tag{5}$$

where $\tilde{x}_j$ is defined in Eqn. 6. For all the outgoing neighbours j of u, $R_{u,j}$ will be constructed as shown in Eq. 4. For all the incoming neighbours d of u, $R_{d,u}$ will be constructed as shown in Eq. 3. The packets constructed for vertex $u \in \tilde{V} \setminus \{s, r\}$ is $m_u = (\{R_{d,u} : \forall d \in \tilde{V}, (d,u) \in \mathbb{PC}\}, t_u, x_u, \{(R_{u,j}, t_{u,j}) : \forall j \in \tilde{V}, (u,j) \in \mathbb{PC}\}, X_u)$. Receiver vertex r receives the following information - $m_r = (\{R_{d,r} : \forall d \in \tilde{V}, (d,r) \in \mathbb{PC}\}, t_r, x_r, \phi, X_r)$. Sender nodes U_0 uses an anonymous secure communication channel to transfer the packets to each of the members $U_i \in \mathbb{PC}, 1 \leq i \leq n$.

Zero-Knowledge Proof is not used in the Setup Phase. In [1], a zero-knowledge proof associated to incoming HTLC and outgoing HTLC was send in each packet meant for the intermediate user. This was needed to verify that an intermediate party can definitely construct solution for its incoming contract if all its outgoing contract were satisfied upon releasing their solution. Constructing zero-knowledge proof is a cumbersome process and might slow down the performance of Layer 2 protocol. For our construction, an intermediate node verifies the relationship using Eq. 9 with the secret share of the node and the commitment values, used in both incoming and outgoing time-locked contracts, as inputs. In this way the computation overhead of sender gets reduced.

5.5.3 TimeLockContractCreate Phase

Any node u, except $u = U_0$, waits for incoming contract to be formed. Any contract formed between u and any of its incoming neighbour, say d, is of the form $contract(d, u, R_{d,u}, t_{d,u})$. It also obtains a value $\tilde{x}_d$ where

$$\begin{aligned} \tilde{x}_d &= x_d + \Sigma_{(j,d)\in\mathbb{PC}} \tilde{x}_j, d \in \tilde{V} \setminus \{s\} \\ \tilde{x}_d &= x_s, d = s \end{aligned} \tag{6}$$

Given the information $m_u = (\{R_{d,u} : \forall d \in \tilde{V}, (d,u) \in \mathbb{PC}\}, t_u, x_u, \{(R_{u,j}, t_{u,j}) : \forall j \in \tilde{V}, (u,j) \in \mathbb{PC}\}, X_u)$ for vertex u, it constructs the value R_u where

$$R_u = \Sigma_{(u,j)\in\mathbb{PC}, j\in\tilde{V}} R_{u,j} \tag{7}$$

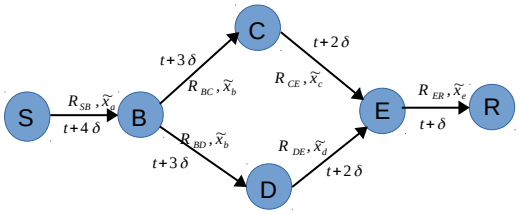


Figure 7: SplitPayLock : TimeLockContractCreate Phase

and checks the following -

$$t_u \stackrel{?}{=} max\{t_{u,j} : (u,j) \in \mathbb{PC}, j \in \tilde{V}\} + \Delta \tag{8}$$

For all outgoing neighbours j of u, it constructs $\tilde{x}_u$ as defined in Eq. 6. For all incoming neighbours d of u,

$$\begin{aligned} e_{d,u} &= \mathcal{H}(x_u\mathcal{G} + R_u || id_{d,u}) \\ R_{d,u} &\stackrel{?}{=} (x_u + e_{d,u}.sk_u)\mathcal{G} + R_u \\ X_u &\stackrel{?}{=} \tilde{x}_u\mathcal{G} \end{aligned} \tag{9}$$

If all the equations hold true, then u sends value $\tilde{x}_u$ to j as well as forms the contract $contract(u, j, R_{u,j}, t_{u,j})$.

5.5.4 TimeLockContractRelease Phase

The release phase starts from the receiver node $U_n = r$ where it checks whether it can construct the discrete log of X_r

$$\begin{aligned} \tilde{x}_r &= x_r + \Sigma_{(u,r)\in\mathbb{PC}:u\in\tilde{V}}\tilde{x}_u) \\ X_r &\stackrel{?}{=} \tilde{x}_r\mathcal{G} \end{aligned} \tag{10}$$

This shows that the key required by r to claim the money is indirectly dependent on the participation of all the channels present in the route $\mathbb{PC}$. If any of the node deviates from the protocol then the payment will fail. It then proceeds to release the condition for all the incoming contracts in order to claim the money.

We provide a generic procedure of the release phase followed by a node $u \in \tilde{V} \setminus \{s\}$ for its incoming contract $R_{w,u}, (w,u) \in \mathbb{PC}$ formed with node w. It constructs

$$s_{w,u} = \begin{cases} \tilde{x_u}, u = r \\ x_u + \Sigma_{(u,v)\in\mathbb{PC}:v\in\tilde{V}} r_{u,v}, u \in \tilde{V} \setminus \{s,r\} \\ e_{w,u} = \mathcal{H}(s_{w,u}\mathcal{G} || id_{w,u}) \\ r_{w,u} = s_{w,u} + e_{w,u}sk_u \end{cases} \tag{11}$$

The value $r_{w,u}$ is revealed to node w.

5.5.5 Example

Consider the subgraph in PCN as shown in Fig. 8. For a transaction of 10 Satoshi from S to R via the payment channels SB, BC, CE, BD, DE and ER. 5 Satoshi each is send via path $S \to B \to C \to$

$E \to R$ and $S \to B \to D \to E \to R$. Starting from the receiver vertex R, with outdegree 0, a random timeout period t_0 is selected. Observe the back edge (R, E). The contract from E to R is assigned a timeout period of t_0. Once the outgoing vertex of E is explored, E is now the vertex with no outgoing edge with unlocked funds. So we observe the back edges EC and ED. Assigning the contract from C to E with a timeout period of $t_0+\Delta$, Δ being some positive value. The same timeout period is assigned to the time-locked contract established on channel DE. Now both C and D have their outgoing edge's funds locked. So contracts to be formed from B to C and from B to D is assigned a timeout period of $t_0 + 2\Delta$. Finally, a timeout period of $t_0 + 3\Delta$ is assigned to the edge SB.

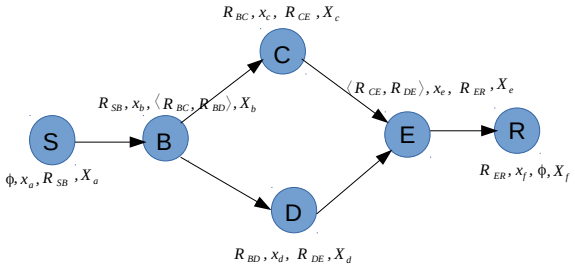


Figure 8: SplitPayLock : Setup

Setup Phase - Following the timeout sequence assigned for each payment channel, the set $\mathbb{PC} = \{RE, EC, ED, BC, BD, SB\}$ is constructed. We discuss how sender executes with the setup phase on the nodes present in $\mathbb{PC}$.

- Consider an elliptic curve group with generator $\mathcal{G}$, with $|\mathcal{G}| = q$, sample 6 independent strings $x_a, x_b, x_c, x_d, x_e, x_f$ for nodes S, B, C, D, E and R respectively.
- For payment channel ER, S constructs the value using the set $\mathbb{PC}$

$$\begin{aligned} \tilde{x_f} &= x_f + 2(x_a + x_b) + x_c + x_d + x_e \\ e_{E,R} &= \mathcal{H}(\tilde{x_f}.\mathcal{G}||id_{E,R}) \\ R_{E,R} &= \tilde{x_f}\mathcal{G} + e_{E,R}.pk_R \end{aligned} \tag{12}$$

  It constructs $X_f = (x_f + 2(x_a + x_b) + x_c + x_d + x_e)\mathcal{G}$ for node R.

- For payment channel CE and DE, S had sampled x_e for E. It now uses $R_{E,R}$ to construct $e_{C,E}$

$$\begin{aligned} e_{C,E} &= \mathcal{H}((x_e.\mathcal{G} + R_{E,R})||id_{C,E}) \\ R_{C,E} &= x_e\mathcal{G} + e_{C,E}.pk_E + R_{E,R} \end{aligned} \tag{13}$$

  It constructs $X_e = (2(x_a + x_b) + x_c + x_d + x_e)\mathcal{G}$ for node E.

$$\begin{aligned} e_{D,E} &= \mathcal{H}((x_e.\mathcal{G} + R_{E,R})||id_{D,E}) \\ R_{D,E} &= x_e\mathcal{G} + e_{D,E}.pk_E + R_{E,R} \end{aligned} \tag{14}$$

- For payment BC, S had sampled x_c for C. It now uses $R_{C,E}$ to construct $e_{B,C}$

$$\begin{aligned} e_{B,C} &= \mathcal{H}((x_c.\mathcal{G} + R_{C,E})||id_{B,C}) \\ R_{B,C} &= x_c\mathcal{G} + e_{B,C}.pk_C + R_{C,E} \end{aligned} \tag{15}$$

It constructs $X_c = (x_a + x_b + x_c)\mathcal{G}$ for node C.

- For payment BD, S had sampled x_d for D. It now uses $R_{D,E}$ to construct $e_{B,D}$

$$\begin{aligned} e_{B,D} &= \mathcal{H}((x_d.\mathcal{G} + R_{D,E})||id_{B,D}) \\ R_{B,D} &= x_d\mathcal{G} + e_{B,D}.pk_D + R_{D,E} \end{aligned} \tag{16}$$

It constructs $X_d = (x_a + x_b + x_d)\mathcal{G}$ for node D.

- For payment SB, S had sampled x_b for B. It now uses $R_{B,C}$ and $R_{B,D}$ to construct $e_{S,B}$

$$\begin{aligned} e_{S,B} &= \mathcal{H}((x_b.\mathcal{G} + R_{B,C} + R_{B,D})||id_{S,B}) \\ R_{S,B} &= x_b\mathcal{G} + e_{S,B}.pk_B + R_{B,C} + R_{B,D} \end{aligned} \tag{17}$$

It constructs $X_b = (x_a + x_b)\mathcal{G}$ for node B and $X_a = x_a$ for node S.

- S forwards the packet $(R_{SB}, x_b, \langle R_{BC}, R_{BD}\rangle, X_b)$ to B, $(R_{BC}, x_c, R_{CE}, X_c)$ to C, $(R_{BD}, x_d, R_{DE}, X_d)$ to D, $(\langle R_{CE}, R_{DE}\rangle, x_e, R_{ER}, X_e)$ to E and (R_{ER}, x_f, ϕ, X_f) to R.

TimeLockContractCreate Phase -

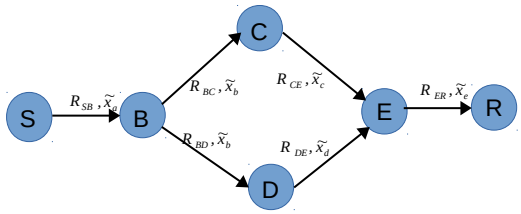


Figure 9: SplitPayLock : TimeLockContractCreate Phase

- S sends $\tilde{x}_a = x_a$ and locks funds with B forming $contract(S, B, R_{SB}, v_1, t_0 + 3\Delta)$, v_1 is the amount B can claim after supplying the key within the given timeout period.

- B checks the following

$$\begin{aligned} e_{S,B} &= \mathcal{H}((x_b.\mathcal{G} + R_{B,C} + R_{B,D})||id_{S,B}) \\ R_{S,B} &\stackrel{?}{=} x_b\mathcal{G} + e_{S,B}.pk_B + R_{B,C} + R_{B,D} \\ X_b &\stackrel{?}{=} (\tilde{x}_a + x_b)\mathcal{G} \end{aligned} \tag{18}$$

If the equality holds true, then B computes $\tilde{x}_b = \tilde{x}_a + x_b$. Next it locks funds with C sending $\tilde{x}_b, contract(B, C, R_{BC}, v_2, t_0 + 2\Delta)$ and locks funds with D sending $\tilde{x}_b, contract(B, D, R_{BD}, v_3, t_0 + 2\Delta)$, where v_2 and v_3 is the amount C and D can claim after supplying the key within the given timeout period.

- C checks the following

$$
\begin{aligned}
e_{B,C} &= \mathcal{H}((x_c.\mathcal{G} + R_{C,E})||id_{B,C}) \\
R_{B,C} &\stackrel{?}{=} x_c\mathcal{G} + e_{B,C}.pk_C + R_{C,E} \\
X_c &\stackrel{?}{=} (\tilde{x}_b + x_c)\mathcal{G}
\end{aligned} \tag{19}
$$

If the equality holds true, then C computes $\tilde{x}_c = \tilde{x}_b + x_c$ and locks funds with E sending $\tilde{x}_c, contract(C, E, R_{CE}, v_4, t_0 + \Delta, \tilde{x}_c)$, where v_4 is the amount E can claim after supplying the key within the given timeout period.

- D checks the following

$$
\begin{aligned}
e_{B,D} &= \mathcal{H}((x_d.\mathcal{G} + R_{D,E})||id_{B,D}) \\
R_{B,D} &\stackrel{?}{=} x_d\mathcal{G} + e_{B,D}.pk_D + R_{D,E} \\
X_d &\stackrel{?}{=} (\tilde{x}_b + x_d)\mathcal{G}
\end{aligned} \tag{20}
$$

If the equality holds true, then D computes $\tilde{x}_d = x_d + \tilde{x}_b$. Next it locks funds with E sending $\tilde{x}_d, contract(D, E, R_{DE}, v_5, t_0 + \Delta)$, where v_5 is the amount E can claim after supplying the key within the given timeout period.

- E checks the following

$$
\begin{aligned}
e_{C,E} &= \mathcal{H}((x_e.\mathcal{G} + R_{E,R})||id_{C,E}) \\
R_{C,E} &\stackrel{?}{=} x_e\mathcal{G} + e_{C,E}.pk_E + R_{E,R} \\
e_{D,E} &= \mathcal{H}((x_e.\mathcal{G} + R_{E,R})||id_{D,E}) \\
R_{D,E} &\stackrel{?}{=} x_e\mathcal{G} + e_{D,E}.pk_E + R_{E,R} \\
X_e &\stackrel{?}{=} (\tilde{x}_c + \tilde{x}_d + x_e)\mathcal{G}
\end{aligned} \tag{21}
$$

If the equality holds true, then E computes $\tilde{x}_e = \tilde{x}_c + \tilde{x}_d + x_e$ and locks funds with R sending $\tilde{x}_e, contract(E, R, R_{ER}, v_6, t_0)$, where v_6 is the amount R can claim after supplying the key within the given timeout period.

TimeLockContractRelease Phase -

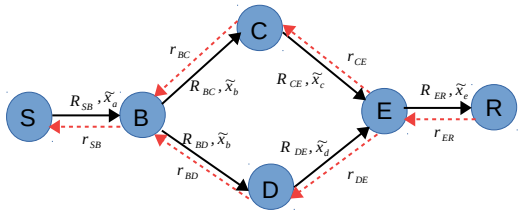


Figure 10: SplitPayLock : TimeLockContractRelease Phase

- R initiates the release phase by constructing r_{ER} in the following way -

$$
\begin{aligned}
\tilde{x}_f &= x_f + \tilde{x}_e \\
e_{E,R} &= \mathcal{H}(\tilde{x}_f.\mathcal{G}||id_{E,R}) \\
r_{ER} &= \tilde{x}_f + e_{E,R}sk_R
\end{aligned} \tag{22}
$$

It sends r_{ER} to E.

- E checks

$$R_{ER} \stackrel{?}{=} r_{ER}\mathcal{G} \tag{23}$$

If this holds true then it constructs r_{CE} and r_{DE} and releases it to C and D respectively. Else it aborts

$$\begin{aligned} e_{C,E} &= \mathcal{H}((x_e + r_{ER}).\mathcal{G}||id_{C,E}) \\ r_{CE} &= x_e + e_{C,E}sk_E + r_{ER} \end{aligned} \tag{24}$$

$$\begin{aligned} e_{D,E} &= \mathcal{H}((x_e + r_{ER}).\mathcal{G}||id_{D,E}) \\ r_{DE} &= x_e + e_{D,E}sk_E + r_{ER} \end{aligned} \tag{25}$$

E sends r_{CE} to C and r_{DE} to D. In a similar fashion, nodes C, D and B construct the solution to claim money from their incoming contract.

- C checks

$$R_{CE} \stackrel{?}{=} r_{CE}\mathcal{G} \tag{26}$$

If this holds true then it constructs r_{BC} and releases it to B. Else aborts.

$$\begin{aligned} e_{B,C} &= \mathcal{H}((x_c + r_{CE}).\mathcal{G}||id_{B,C}) \\ r_{BC} &= x_c + e_{B,C}sk_C + r_{CE} \end{aligned} \tag{27}$$

It sends r_{BC} to B.

- D checks

$$R_{DE} \stackrel{?}{=} r_{DE}\mathcal{G} \tag{28}$$

If this holds true then it constructs r_{BD} and releases it to B. Else it aborts.

$$\begin{aligned} e_{B,D} &= \mathcal{H}((x_d + r_{DE}).\mathcal{G}||id_{B,D}) \\ r_{BD} &= x_d + e_{B,D}sk_D + r_{DE} \end{aligned} \tag{29}$$

It sends r_{BD} to B.

- B checks

$$\begin{aligned} R_{BC} &\stackrel{?}{=} r_{BC}\mathcal{G} \\ R_{BD} &\stackrel{?}{=} r_{BD}\mathcal{G} \end{aligned} \tag{30}$$

If this holds true then it constructs r_{SB} and releases it to S. Else it aborts.

$$\begin{aligned} e_{S,B} &= \mathcal{H}((x_b + r_{BC} + r_{BD}).\mathcal{G}||id_{S,B}) \\ r_{SB} &= x_b + e_{S,B}sk_B + r_{BC} + r_{BD} \end{aligned} \tag{31}$$

- S checks

$$R_{SB} \stackrel{?}{=} r_{SB}\mathcal{G} \tag{32}$$

Claim 1 *If all the participants $u \in \tilde{V}$ involved in routing the payment (s, r, val) behave honestly, then payment protocol achieves strong atomicity.*

Proof. If all the payment channels present in the set $\mathbb{PC}$ execute *TimeLockContractCreate* Phase and each party have shared their partial secret value with their respective counterparties, then by our proposed construction receiver can reconstruct the key. Once the receiver has constructed the key, it triggers *TimeLocContracRelease* Phase and claims the fund from their incoming contract within the given timeout period of the contract. Iteratively, the incoming neighbors of the receiver construct their solution for claiming their fund to retain consistency property. Let us assume that $\exists v \in \mathbb{PC}$ which is unable to claim funds within the timeout period stated in the contract. All the channels present on this path starting from sender s up to the node v roll back to their original state after the timeout period. This means either v misbehaves and denies service by going offline or it obtained a wrong solution from its outgoing neighbors. The former situation can be ruled out since all the parties are considered to act honestly. If we consider the latter situation, it implies that its outgoing neighbor has deviated from the protocol. This contradicts our assumption. Hence if the receiver has received the solution and claimed its fund then all the parties update their channel state and claim their fund as well.

Claim 2 *If a participant $u \in \tilde{V} \setminus \{s, r\}$ involved in routing the payment (s, r, val) behave dishonestly, then payment protocol achieves weak atomicity.*

Proof. The protocol proceeds in a similar way as stated in Claim 1 assuming all the parties behaved honestly till it is u's turn to execute *TimeLockContractRelease* phase. If u intentionally goes offline or provides a wrong solution then it cannot claim the money from the incoming contract. All the nodes on the path starting from s till u wait till the timeout period to withdraw funds locked in the contract. Sender s ends up making a partial payment.

5.6 Privacy Analysis of SplitPayLock

Theorem 2 *If Discrete Logarithm problem is hard and function $\mathcal{H}$ is random oracle, then the protocol* SplitPayLock UC*-realizes the ideal functionality $\mathcal{F}$ in the $(\mathcal{F}_{syn}, \mathcal{F}_{smt}, \mathcal{F}_{anon}$)-hybrid model.*

In order to prove Theorem 2, the ideal world simulator Sim needs to ensure the output of execution of the an instance of the protocol *SplitPayLock* in $(\mathcal{F}_{syn}, \mathcal{F}_{smt}, \mathcal{F}_{anon})$-hybrid world is indistinguishable as that in the ideal world, even in presence of corrupt parties. We consider here the following cases for basic PCN operations - openChannel (if any one party is malicious), closeChannel(if any one of the parties is malicious) and payment (sender is malicious, receiver is malicious and when an intermediate party is malicious). The environment $\mathcal{Z}$ can use the information leaked by adversary $\mathcal{A}$ or actively influence the execution. It supplies the input to the parties, gets the output and can even corrupt any parties to learn their internal values, control the execution by keeping a tab on the input and output send from that party.

5.6.1 OpenPaymentChannel$(U_i, U_j, \alpha, t, fee)$ in Ideal World

Given a payment channel between user U_i and U_j, with U_i initiating the request for opening a channel, with balance α, timeout value t and fee fee is the fee charged on using the channel.

U_i **is corrupted** On corruption of user U_i by adversary $\mathcal{A}$, a channel open request is send to the other party. Upon agreement of both the parties, the protocol in hybrid world returns channel id $id_{(U_i,U_j)}$ to U_j upon successful execution. Else it returns an abort. Sim will replicate the same in the

ideal world by sending $(sid, open, U_i, U_j, \alpha, t, fee)$ to $\mathcal{F}$ and outputs $(sid, opened, id_{(U_i,U_j)}, U_i, \alpha, t, fee)$ to U_j upon success.

U_j **is corrupted** $\mathcal{A}$ corrupts U_j and a channel open request initiated by honest party U_i. Upon agreement of both the parties, the protocol in hybrid world returns channel id $id_{(U_i,U_j)}$ if the execution is successful. Else it will abort. *Sim* will replicate the same in the ideal world by sending $(sid, ack - close, id_{(U_i,U_j)}, U_i, t, m)$ to $\mathcal{F}$ and then it returns (sid, h), where h is the event identifier.

5.6.2 ClosePaymentChannel($id_{(U_i,U_j)}, h$) in Ideal World

Given an exisiting channel $id_{(U_i,U_j)}$ between U_i and U_j with event id h, with U_i initiating the request.

U_i **is corrupted** On receiving a request from the corrupted party U_i, *Sim* will check if such a channel exists in B. If it exists, then it will extract the event identifier h and send request $(sid, close, id_{(U_i,U_j)}, h)$ to $\mathcal{F}$. Else it will abort.

U_j **is corrupted** If *Sim* receives $(sid, ack - close, sid, h)$ from $\mathcal{F}$, then it continues with the protocol execution.

5.6.3 Payment in Ideal World

`payVal`$(p(s,r), val)$: $p(s,r)$ denotes a path between sender s and receiver r. It is defined by a set of identifiers $id_{(s,v_1)}, id_{(v_1,v_2)}, \ldots, id_{(v_n,r)}$, $s, v_1, v_2, \ldots, v_n, r \in V$, having enough credit to allow transfer of val from s to r. The specification of the protocol differs based on whether the node is a sender, receiver or intermediate party, we discuss each case separately.

Sender S - An adversary $\mathcal{A}$ initiates payment by providing each honest user U_i, the tuple $(\{R_{d,U_i} : \forall d \in \tilde{V}, (d, U_i) \in \mathbb{PC}\}, t_{U_i}, x_{U_i}, v, \{(R_{U_i,j}, t_{U_i,j}) : \forall j \in \tilde{V}, (U_i, j) \in \mathbb{PC}\})$ and $(\{R_{d,U_n} : \forall d \in \tilde{V}, (d, U_n) \in \mathbb{PC}\}, t_{U_n}, x_{U_n})$ to receiver U_n. For each U_i, *Sim* checks if $t_{U_i} \geq t_{U_i,j}, \forall j \in \tilde{V}, (U_i, j) \in \mathbb{PC}$. If there exists a relation between $x_{U_i}, \{R_{d,U_i} : \forall d \in \tilde{V}, (d, U_i) \in \mathbb{PC}\}$ and $\{R_{U_i,j} : \forall j \in \tilde{V}, (U_i, j) \in \mathbb{PC}\}$, then *Sim* sends $(pay, \overrightarrow{U_{in,U_i}}, \overrightarrow{U_{out,U_i}}, v_i, t_{prev}, t_i\}$ to $\mathcal{F}$. For receiver U_n, it will send $(pay, \overrightarrow{U_{in,U_i}}, \perp, v_i, t_{prev}, t_i\}$ to $\mathcal{F}$, provided $R_{d,U_n} = (x_{U_n} + e_{d,U_n}.sk_{U_n})\mathcal{G}, \forall (d, U_n) \in \mathbb{PC}, d \in \tilde{V}$, else it aborts. The simulator confirms the payment for a payment channel (U_i, U_j) only when it receives from the user U_j an $r_{U_i,U_j} = \Sigma_{(U_j,U_l)\in\mathbb{PC}} r_{U_j,U_l}$ such that $R_{U_i,U_j} = (r_{U_i,U_j} + x_{U_j} + e_{U_i,U_j} sk_{U_j})\mathcal{G}$. In case $\mathcal{A}$ finds a r^* for (U_j, U_k) without querying *Sim* such that $R_{U_j,U_k} = r^*\mathcal{G}$ but $r_{U_i,U_j} \neq \Sigma_{(U_j,U_l)\in\mathbb{PC}, U_l \neq U_k} r_{U_j,U_l} + r^*$ then *Sim* aborts. But finding an $r*$ is equivalent to breaking discrete logarithm hardness. Probability of this event is $\frac{1}{q}, q = |\mathbb{G}|$, which is negligible since q is a verly large prime number. If the receiver is honest then *Sim* confirms the payment if the amount v corresponds to what was agreed upon and corresponds to what agreed with the sender, provided $R_{d,U_n} = (x_{U_n} + e_{d,U_n}.sk_{U_n})\mathcal{G}, \forall d \in \tilde{V}, (d, U_n) \in \mathbb{PC}$. If the payment is confirmed for a channel (U_i, U_j), then an entry $(con_{U_i,U_j}, r_{U_i,U_j}, R_{U_i,U_j})$ is added to $\mathcal{C}$ where $R_{U_i,U_j} = r_{U_i,U_j}\mathcal{G}$.

Receiver R *Sim* receives $(\overrightarrow{con_{in,U_n}}, v, t_n, \perp), \forall U_j \in \tilde{V}, (U_j, U_n) \in \mathbb{PC}$ from $\mathcal{F}$, then it samples a random x_n and returns to $\mathcal{A}$ the tuple $(x_n, \overrightarrow{R_{in,U_n}}, v)$. If the adversary returns the same x_n then *Sim* return T to $\mathcal{F}$, otherwise it sends $\perp$.

Intermediate party U_i $\mathcal{F}$ informs Sim about a corrupted user U_i involved in the payment protocol by sending a message $(\overrightarrow{U_{in,U_i}}, \overrightarrow{U_{out,U_i}}, v, t_{U_i}, t_{prev,U_i})$. Sim samples secret random string $r'_{U_i,U_j} \leftarrow \{0,1\}^\lambda : R'_{U_i,U_j} = r'_{U_i,U_j}\mathcal{G}, \forall (U_i, U_j) \in \mathbb{PC}$ and another secret value $x'_{U_i} \leftarrow \{0,1\}^\lambda$. It constructs $R_{U_k,U_i}, \forall (U_k, U_i) \in \mathbb{PC}$ in the following way :

$$R_{U_k,U_i} = x'_{U_i}\mathcal{G} + e_{U_k,U_i}.pk_{U_i} + \Sigma_{(U_i,U_j)\in\mathbb{PC}} R'_{U_i,U_j} \tag{33}$$

It constructs the packet $(\{R_{U_k,U_i} : \forall U_k \in \tilde{V}, (U_k, U_i) \in \mathbb{PC}\}, t_{U_i}, x'_{U_i}, \{(R_{U_i,U_j}, t_{U_i,U_j}) : \forall U_j \in \tilde{V}, (U_i, U_j) \in \mathbb{PC}\})$ and shares it with $\mathcal{A}$. If $\mathcal{A}$ outputs a string x' by itself without querying Sim such that $x'_i = x'_{U_i} + \Sigma_{(U_i,U_j)\in\mathbb{PC}} r'_{U_i,U_j}$ and satisfies the solution needed for $R_{U_k,U_i}, \forall U_k \in \tilde{V}, (U_k, U_i) \in \mathbb{PC}$ then Sim aborts. Probablity of finding such an x'_i is is negligible, given that $\mathcal{A}$ is a probabilistic polynomial time algorithm and finding an x'_i is equivalent to breaking discrete logarithm problem. Probability of this event is $\frac{1}{q}, q = |\mathbb{G}|$, which is negligible since q is a verly large prime number. Thus the probability that Sim aborts is also negligible.

If it is the case that $\mathcal{A}$ obtains $\Sigma_{(U_i,U_j)\in\mathbb{PC}} r'_{U_i,U_j}$ from Sim upon querying in order to construct x'_i, then it is treated as a legitimate event. If $\mathcal{A}$ is able to output $r_{U_k,U_i}, \forall U_k \in \tilde{V}, (U_k, U_i) \in \mathbb{PC}$, then Sim sends T to $\mathcal{F}$ and entries $(con_{U_k,U_i}, r_{U_k,U_i}, R_{U_k,U_i})$ is added to $\mathcal{C}$ where $R_{U_k,U_i} = r_{U_k,U_i}\mathcal{G}$. In case of error, it sends $\perp$.

Computational Indistinguishability of the real world from ideal world Considering the Sim to be a polynomially bounded algorithm, the view of the environment differs from the real world and the ideal world only when Sim aborts. This is because the execution does not stop in the real world even in the presence of corrupted parties. We have to argue that the probability of Sim aborting is negligible. For the two operations, *openPaymentChannel* and *closePaymentChannel*, the behavior of $\mathcal{A}$ is not different from Sim. Hence the argument is trivial. For the payment protocol, we have argued considering each of the cases - sender is corrupted, receiver is corrupted and an intermediate party is corrupted. The payment protocol does not stop at an honest node and Sim works identically without aborting. However Sim aborts in case $\mathcal{A}$ can find the discrete log value to release the time-locked contract without querying Sim. In that case, Sim will abort. But as shown, this can happen with negligible probability as it is equivalent to solving discrete logarithm problem.

6 Performance Analysis

6.1 Experimental Setup

In this section, we define the experimental setup. The code for *SplitPayLock* is available in [27]. System configuration used is : `Intel Core i5-8250U CPU, Kabylake GT2 octa core processor, frequency 1.60 GHz`, OS : *Ubuntu-18.04.1 LTS* (64 bit). The programming language used is C, compiler - gcc version 5.4.0 20160609. The library *igraph* was used for generating random graphs of size ranging from 50 to 25000, based on Barábasi-Albert model [51], [52]. Payment Channel Network follows the scale free network where certain nodes function as hub (like central banks), having higher degree compared to other nodes [53]. For implementing the cryptographic primitives in both *SplitPayLock* and *Multihop HTLC*, we use the library *OpenSSL*, version-1.0.2 [54] and SHA-256 has been modeled as a random oracle. For constructing the zero-knowledge proof for *Multihop HTLC*, we have used C-based implementation of ZKBoo[55]. The number rounds for running the protocol is set to 136,

which guarantees soundness error of 2^{-80} for the proof and witness length is set to 32 bytes. For implementing *SplitPayLock*, we have considered the elliptic curve secp224r1.

6.2 Evaluation

Figure 11: Time taken for Payment

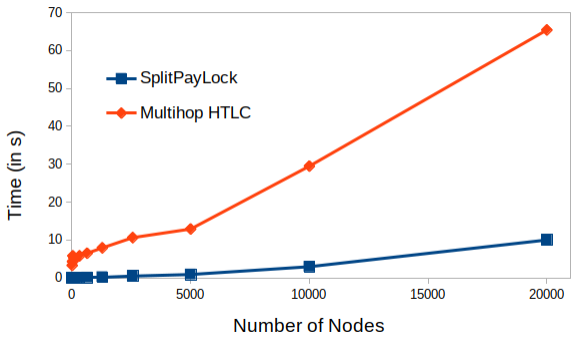
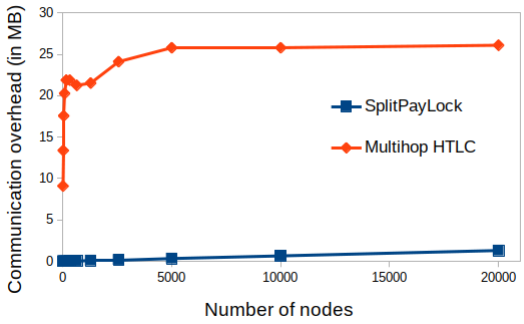


Figure 12: Communication overhead



Following metrics are used to compare the performance of the payment protocol, *SplitPayLock* with *Multi-Hop HTLC* [1]:

- TTP (*Time taken for payment*) : Given set of paths for payment transfer, it is the time taken for construction of hashed time-lock contract across all the edges in the path and completion of payment upon successfully fulfilling the criteria set in the contract.
- Communication Overhead: For the given payment protocol, the number of message packets exchanged between the nodes in terms of *bytes.*

For the graph given in Fig. 10, in order to transfer a value of 10 Satoshi from S to R, two paths $S \to B \to C \to E \to R$ and $S \to B \to D \to E \to R$ are obtained, each carrying 5 Satoshi. Multihop

HTLC is applied to each path, one at a time. The time taken to complete the payment protocol is 1.53s and communication overhead is 6.483 MB, considering each path having 5 users each. On executing *SplitPayLock* for the same payment, the execution time taken is 1.9ms and communication overhead is 1.087 KB.

Testing on Real Instances. We test our protocol on a Ripple network [15], comprising around 20000 nodes. As stated in Malavolta et al. [1], due to the absence of widespread PCN in practice, we set the path length for each payment as has been suggested for the Ripple credit network. Our proposed payment protocol takes around 10s to complete the payment with a communication overhead of less than 1.5 MB. Multihop HTLC takes around 65s to complete the protocol and incurs a communication overhead of 26 MB. Considering an instance of Lightning Network as stated in [26], comprising 2500 nodes. It takes around 485ms and communication overhead of 0.16 MB as compared to 10.6s and communication overhead of 24 MB by Multihop HTLC.

Testing on Simulated Instances. We consider synthetically generated graphs, with the number of nodes ranging from 10 to 20000. We vary the source-sink pair and use value 40 for transfer from payer to a payee for all the instances. Apart from simulated graphs, Overall, the result demonstrates the benefit of considering all the split simultaneously instead of one path at a time in terms of scalability and efficiency in terms of computation cost and resource utilization.

7 Conclusion

In this paper, we have proposed a novel privacy-preserving multipath off-chain payment protocol for Payment Channel Network, *SplitPayLock*. This protocol guarantees strong atomicity, when all parties involved in the routing of the transaction act honestly, and weak atomicity, even if a single party deviates from the protocol. We analysed the performance of the protocol on some real instances like Lightning Network and Ripple-like Network. From the results, it was inferred that our proposed payment protocol has less execution time and low communication overhead as compared to existing payment protocols like Multihop HTLC [1]. It is efficient and scalable as the setup phase doesn't require any complex computation. Our protocol instance has been defined for a transaction between a payer and payee but it can be extended to handle multiple transactions by enforcing blocking protocol or non-blocking protocol to resolve deadlocks in concurrent payments [1]. As future work, we will extend our protocol to scriptless script-based two-party locking protocol as stated in Malavolta et al.[12].

8 Acknoledgement

The authors thank Prabal Banerjee, *Indian Statistical Institute Kolkata* and Ram Goving Singh, *ICERT, Ministry of Electronics and Information Technology* for fruitful discussions and invaluable technical support.

References

[1] G. Malavolta, P. Moreno-Sanchez, A. Kate, M. Maffei, and S. Ravi, "Concurrency and privacy with payment-channel networks," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer*

and Communications Security, pp. 455–471, ACM, 2017.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[3] K. J. O'Dwyer and D. Malone, "Bitcoin mining and its energy footprint," 2014.

[4] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "Consensus in the age of blockchains," *arXiv preprint arXiv:1711.03936*, 2017.

[5] S. King and S. Nadal, "Ppcoin: Peer-to-peer crypto-currency with proof-of-stake," *self-published paper, August*, vol. 19, 2012.

[6] W. Li, S. Andreina, J.-M. Bohli, and G. Karame, "Securing proof-of-stake blockchain protocols," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pp. 297–315, Springer, 2017.

[7] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, *et al.*, "On scaling decentralized blockchains," in *International Conference on Financial Cryptography and Data Security*, pp. 106–125, Springer, 2016.

[8] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," *See https://lightning. network/lightning-network-paper. pdf*, 2016.

[9] C. Decker, R. Russell, and O. Osuntokun, "eltoo: A simple layer2 protocol for bitcoin," *White paper: https://blockstream. com/eltoo. pdf*, 2018.

[10] C. Decker and R. Wattenhofer, "A fast and scalable payment network with bitcoin duplex micropayment channels," in *Symposium on Self-Stabilizing Systems*, pp. 3–18, Springer, 2015.

[11] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 17–30, ACM, 2016.

[12] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei, "Multi-hop locks for secure, privacy-preserving and interoperable payment-channel networks," in *Network and Distributed System Security Symposium*, 2019.

[13] E. Rohrer, J.-F. Laß, and F. Tschorsch, "Towards a concurrent and distributed route selection for payment channel networks," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pp. 411–419, Springer, 2017.

[14] P. Prihodko, S. Zhigulin, M. Sahno, A. Ostrovskiy, and O. Osuntokun, "Flare: An approach to routing in lightning network," *White Paper (bitfury.com/content/5-white-papers-research/whitepaper
flare an_ approach_ to_ routing_ in_ lightning_ n etwork_ 7_ 7_ 2016. pdf)*, 2016.

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

[16] S. Roos, P. Moreno-Sanchez, A. Kate, and I. Goldberg, "Settling payments fast and private: Efficient decentralized routing for path-based transactions," in *Network and Distributed System Security Symposium*, 2018.

[17] B. Viswanath, M. Mondal, K. P. Gummadi, A. Mislove, and A. Post, "Canal: Scaling social network-based sybil tolerance schemes," in *Proceedings of the 7th ACM european conference on Computer Systems*, pp. 309–322, ACM, 2012.

[18] R. Yu, G. Xue, V. T. Kilari, D. Yang, and J. Tang, "Coinexpress: A fast payment routing mechanism in blockchain-based payment channel networks," in *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, pp. 1–9, IEEE, 2018.

[19] P. Hoenisch and I. Weber, "Aodv–based routing for payment channel networks," in *International Conference on Blockchain*, pp. 107–124, Springer, 2018.

[20] P. Moreno-Sanchez, A. Kate, M. Maffei, and K. Pecina, "Privacy preserving payments in credit networks," in *Network and Distributed Security Symposium*, 2015.

[21] M. Green and I. Miers, "Bolt: Anonymous payment channels for decentralized currencies," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 473–489, ACM, 2017.

[22] A. Miller, I. Bentov, R. Kumaresan, and P. McCorry, "Sprites: Payment channels that go faster than lightning," in *Twenty-Third International Conference on Financial Cryptography and Data Security 2019*, 2019.

[23] C. Egger, P. Moreno-Sanchez, and M. Maffei, "Atomic multi-channel updates with constant collateral in bitcoin-compatible payment-channel networks," in *26th ACM Conference on Computer and Communications Security*, ACM, 2019.

[24] "Amp: Atomic multi-path payments over lightning.." `https://lists.linuxfoundation.org/pipermail/lightning-2018-February/000993.html`, February 2018.

[25] V. Bagaria, J. Neu, and D. Tse, "Boomerang: Redundancy improves latency and throughput in payment networks," *arXiv preprint arXiv:1910.01834*, 2019.

[26] I. A. Seres, L. Gulyás, D. A. Nagy, and P. Burcsi, "Topological analysis of bitcoin's lightning network," *arXiv preprint arXiv:1901.04972*, 2019.

[27] "Splipaylock." `https://www.dropbox.com/sh/x9pngj005dxh87b/AAAJNt-WquV0JZTspnijEXNVa?dl=0`, 2019.

[28] "Raiden network." `http://raiden.network/`, July 2017.

[29] S. Thomas and E. Schwartz, "A protocol for interledger payments," *URL https://interledger.org/interledger.pdf*, 2015.

[30] "Atomic cross-chain trading.." `https://en.bitcoin.it/wiki/ Atomic_cross-chain_trading`, July 2017.

[31] J. Lind, I. Eyal, F. Kelbert, O. Naor, P. Pietzuch, and E. G. Sirer, "Teechain: Scalable blockchain payments using trusted execution environments," *arXiv preprint arXiv:1707.05454*, 2017.

[32] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "Perun: Virtual payment hubs over cryptographic currencies," tech. rep., IACR Cryptology ePrint Archive 2017, 2017.

[33] S. Dziembowski, S. Faust, and K. Hostáková, "General state channel networks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 949–966, ACM, 2018.

[34] V. Sivaraman, S. B. Venkatakrishnan, M. Alizadeh, G. Fanti, and P. Viswanath, "Routing cryptocurrency with the spider network," *arXiv preprint arXiv:1809.05088*, 2018.

[35] S. Roos, M. Beck, and T. Strufe, "Voute-virtual overlays using tree embeddings," *arXiv preprint arXiv:1601.06119*, 2016.

[36] C. Grunspan and R. Pérez-Marco, "Ant routing algorithm for the lightning network," *arXiv preprint arXiv:1807.00151*, 2018.

[37] K. Atlas, "The inevitability of privacy in lightning networks, 2017," *URL https://www. kristovatlas. com/the-inevitability-of-privacy-in-lightning-networks/. [Online.*

[38] J. Herrera-Joancomartí and C. Pérez-Solà, "Privacy in bitcoin transactions: new challenges from blockchain scalability solutions," in *Modeling Decisions for Artificial Intelligence*, pp. 26–44, Springer, 2016.

[39] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais, "Sok: Off the chain transactions," *IACR Cryptology ePrint Archive*, vol. 2019, p. 360, 2019.

[40] E. Heilman, L. Alshenibr, F. Baldimtsi, A. Scafuro, and S. Goldberg, "Tumblebit: An untrusted bitcoin-compatible anonymous payment hub," in *Network and Distributed System Security Symposium*, 2017.

[41] I. Giacomelli, J. Madsen, and C. Orlandi, "Zkboo: Faster zero-knowledge for boolean circuits.," in *USENIX Security Symposium*, pp. 1069–1083, 2016.

[42] B. Yu, S. K. Kermanshahi, A. Sakzad, and S. Nepal, "Chameleon hash time-lock contract for privacy preserving payment channel networks," in *International Conference on Provable Security*, pp. 303–318, Springer, 2019.

[43] E. Rohrer, J. Malliaris, and F. Tschorsch, "Discharged payment channels: Quantifying the lightning network's resilience to topology-based attacks," in *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pp. 347–356, IEEE, 2019.

[44] M. Herlihy, "Atomic cross-chain swaps," in *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pp. 245–254, ACM, 2018.

[45] S. Dziembowski and P. Kędzior, "Ethna: Channel network with dynamic internal payment splitting,"

[46] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *Proceedings of the 1st ACM conference on Computer and communications security*, pp. 62–73, ACM, 1993.

[47] S. D. Galbraith and P. Gaudry, "Recent progress on the elliptic curve discrete logarithm problem," *Designs, Codes and Cryptography*, vol. 78, no. 1, pp. 51–72, 2016.

[48] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*, pp. 136–145, IEEE, 2001.

[49] J. Camenisch and A. Lysyanskaya, "A formal treatment of onion routing," in *Annual International Cryptology Conference*, pp. 169–187, Springer, 2005.

[50] S. Dziembowski, L. Eckey, and S. Faust, "Fairswap: How to fairly exchange digital goods," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 967–984, ACM, 2018.

[51] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks," *Reviews of modern physics*, vol. 74, no. 1, p. 47, 2002.

[52] A.-L. Barabási and E. Bonabeau, "Scale-free networks," *Scientific american*, vol. 288, no. 5, pp. 60–69, 2003.

[53] M. A. Javarone and C. S. Wright, "From bitcoin to bitcoin cash: a network analysis," in *Proceedings of the 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems*, pp. 77–81, ACM, 2018.

[54] The OpenSSL Project, "OpenSSL: The open source toolkit for SSL/TLS." `www.openssl.org`, 10 September, 2019.

[55] "Source code : C based implementation of zkboo." `https://github.com/Sobuno/ZKBoo/`, October, 2016.