Concurrency and Privacy with Payment-Channel Networks*

Giulio Malavolta†
Friedrich-Alexander-University Erlangen-Nürnberg
malavolta@cs.fau.de

Pedro Moreno-Sanchez†
Purdue University
pmorenos@purdue.edu

Aniket Kate
Purdue University
aniket@purdue.edu

Matteo Maffei
TU Wien
matteo.maffei@tuwien.ac.at

Srivatsan Ravi
University of Southern California
srivatsr@usc.edu

Abstract

Permissionless blockchains protocols such as Bitcoin are inherently limited in transaction throughput and latency. Current efforts to address this key issue focus on off-chain payment channels that can be combined in a Payment-Channel Network (PCN) to enable an unlimited number of payments without requiring to access the blockchain other than to register the initial and final capacity of each channel. While this approach paves the way for low latency and high throughput of payments, its deployment in practice raises several privacy concerns as well as technical challenges related to the inherently concurrent nature of payments that have not been sufficiently studied so far.

In this work, we lay the foundations for privacy and concurrency in PCNs, presenting a formal definition in the Universal Composability framework as well as practical and provably secure solutions. In particular, we present Fulgor and Rayo. Fulgor is the first payment protocol for PCNs that provides provable privacy guarantees for PCNs and is fully compatible with the Bitcoin scripting system. However, Fulgor is a blocking protocol and therefore prone to deadlocks of concurrent payments as in currently available PCNs. Instead, Rayo is the first protocol for PCNs that enforces *non-blocking progress* (i.e., at least one of the concurrent payments terminates). We show through a new impossibility result that non-blocking progress necessarily comes at the cost of weaker privacy. At the core of Fulgor and Rayo is Multi-Hop HTLC, a new smart contract, compatible with the Bitcoin scripting system, that provides conditional payments while reducing running time and communication overhead with respect to previous approaches. Our performance evaluation of Fulgor and Rayo shows that a payment with 10 intermediate users takes as few as 5 seconds, thereby demonstrating their feasibility to be deployed in practice.

1 Introduction

Bitcoin [57] is a fully decentralized digital cryptocurrency network that is widely adopted today as an alternative monetary payment system. Instead of accounting payments in a ledger locally maintained by a trusted financial institute, these are logged in the Bitcoin blockchain, a database replicated among mutually distrusted users around the world who update it by means of a global consensus algorithm based on proof-of-work. Nevertheless, the permissionless nature of this consensus algorithm limits the transaction rate to tens of transactions per second whereas other payment networks such as Visa support peaks of up to 47,000 transactions per second [18].

In the forethought of a growing number of Bitcoin users and most importantly payments about them, scalability is considered today an important concern among the Bitcoin community [67, 3]. Several research and industry efforts are dedicated today to overcome this important burden [3, 62, 60, 32, 4, 2].

The use of Bitcoin *payment channels* [6, 32] to realize off-chain payments has flourished as a promising approach to overcome the Bitcoin scalability issue. In a nutshell, a pair of users open a payment channel by adding a single transaction to the blockchain where they lock their bitcoins in a de-

*This is the revision 6 September 2017. The most recent version is available at `https://eprint.iacr.org/2017/820`

†Both authors contributed equally and are considered to be co-first authors.

posit secured by a Bitcoin smart contract. Several off-chain payments can be then performed by locally agreeing on the new distribution of the deposit balance. Finally, the users sharing the payment channel perform another Bitcoin transaction to add the final balances in the blockchain, effectively closing the payment channel.

In this manner, the blockchain is required to open and close a payment channel but not for any of the (possibly many) payments between users, thereby reducing the load on the blockchain and improving the transaction throughput. However, this simple approach is limited to direct payments between two users sharing an open channel. Interestingly, it is in principle possible to leverage a path of opened payment channels from the sender to the receiver with enough capacity to settle their payments, effectively creating a *payment-channel network (PCN)* [60].

Many challenges must be overcome so that such a PCN caters a wide deployment with a growing number of users and payments. In particular, today we know from similar payment systems such as credit networks [36, 37, 15, 17] that a fully-fledged PCN must offer a solution to several issues, such as liquidity [29, 55], network formation [30], routing scalability [61, 71], concurrency [49], and privacy [56, 54, 49] among others.

The Bitcoin community has started to identify these challenges [47, 41, 40, 48, 43, 22, 3, 67]. Nevertheless, current PCNs are still immature and these challenges require to be thoroughly studied. In this work, we lay the foundations for privacy and concurrency in PCNs. Interestingly, we show that these two properties are connected to each other and that there exists an inherent trade-off between them.

The Privacy Challenge. It seems that payment channels necessarily improve the privacy of Bitcoin payments as they are no longer logged in the blockchain. However, such pervading idea has started to be questioned by the community and it is not clear at this point whether a PCN can offer sufficient privacy guarantees [68, 22, 43]. Recent research works [47, 41, 40] propose privacy preserving protocols for payment hub networks, where all users perform off-chain payments through a unique intermediary. Unfortunately, it is not clear how to extend these solutions to multi-hop PCNs.

Currently, there exist some efforts in order to define a fully-fledged PCN [19, 60, 13, 10]. Among them, the Lightning Network [60] has emerged as the most prominent PCN among the Bitcoin community [1]. However, its current operations do not provide all the privacy guarantees of interest in a PCN. For instance, the computation of the maximum possible value to be routed through a payment path requires that intermediate users reveal the current capacity of their payment channels to the sender [62, Section 3.6], thereby leaking sensitive information. Additionally, the Bitcoin smart-contract used in the Lightning Network to enforce atomicity of updates for payment channels included in the payment path, requires to reveal a common hash value among each user in the path that can be used by intermediate users to derive who is paying to whom [60]. As a matter of fact, while a plethora of academic papers have studied the privacy guarantees offered by current Bitcoin payments on the Bitcoin blockchain [52, 25, 66, 45, 64, 21, 51], there exists at present no rigorous analysis of the privacy guarantees offered by or desirable in PCNs. The lack of rigorous definitions for their protocols, threat model and privacy notions, hinders a formal security and privacy analysis of ongoing attempts, let alone the development of provably secure and privacy-preserving solutions.

The Concurrency Challenge. The consensus algorithm, e.g., proof-of work in Bitcoin, eases the serialization of concurrent on-chain payments. A miner with access to all concurrent payments at a given time can easily serialize them following a set of predefined rules (e.g., sort them by payment fee) before they are added to the blockchain. However, this is no longer the case in a PCN: The bulk of off-chain payments are not added to the blockchain and they cannot be serialized during consensus. Moreover, individual users cannot avoid concurrency issues easily either as a payment might involve several other users apart from payer and payee.

In current PCNs such as the Lightning Network, a payment is aborted as soon as a payment channel in the path does not have enough capacity (possibly allocated for another in-flight payment concurrently). This, however, leads to deadlock (and starvation) situations where none of the in-flight payments terminates. In summary, although concurrent payments are likely to happen when current PCNs scale to a large number of users and off-chain payments, the inherent concurrency issues have not

been thoroughly investigated yet.

Our Contribution. This work makes the following contributions:

First, we formalize for the first time the security and privacy notions of interest for a PCN, namely *balance security*, *value privacy* and *sender/receiver anonymity*, following the universal composability (UC) framework [27].

Second, we study for the first time the concurrency issues in PCNs and present two protocols Fulgor and Rayo that tackle this issue with different strategies. Fulgor is a blocking protocol in line with concurrency solutions proposed in somewhat similar payment networks such as credit networks [49, 15] that can lead to deadlocks where none of the concurrent payments go through. Overcoming this challenge, Rayo is the first protocol for PCNs guaranteeing *non-blocking* progress [42, 20]. In doing so, Rayo ensures that at least one of the concurrent payments terminates.

Third, we characterize an arguably surprising tradeoff between privacy and concurrency in PCNs. In particular, we demonstrate that any PCN that enforces non-blocking progress inevitably reduces the anonymity set for sender and receiver of a payment, thereby weakening the privacy guarantees.

Fourth, we formally describe the Multi-Hop Hash Time-Lock Contract (Multi-Hop HTLC), a smart contract that lies at the core of Fulgor and Rayo and which, in contrast to the Lightning Network, ensures privacy properties even against users in the payment path from payer to payee. We formally define the Multi-Hop HTLC contract and provide an efficient instantiation based on the recently proposed zero-knowledge proof system ZK-Boo [38], that improves on previous proposals [69] by reducing the data required from 650 MB to 17 MB, the running time for the prover from 600 ms to 309 ms and the running time for verifying from 500 ms to 130 ms. Moreover, Multi-Hop HTLC does not require changes to the current Bitcoin scripting system, can thereby be seamlessly deployed in current PCNs, and is thus of independent interest.

Finally, we have implemented a prototype of Fulgor and Rayo in Python and evaluated the running time and communication cost to perform a payment. Our results show that a privacy-preserving payment in a path with 10 intermediate users can be carried out in as few as 5 seconds and incurs on 17 MB of communication overhead. This shows that our protocols for PCN are in line with with other privacy-preserving payment systems [54, 49]. Additionally, our evaluation shows that Fulgor and Rayo can scale to cater a growing number of users with a reasonably small overhead that can be further reduced with an optimized implementation.

Organization. Section 2 overviews the required background. Section 3 defines the problem we tackle in this work and overviews Fulgor and Rayo, our privacy preserving solution for PCNs. Section 4 details the Fulgor protocol. Section 5 describes our study of concurrency in PCNs and details the Rayo protocol. Section 6 describes our implementation and the evaluation results. Section 7 discusses the related work and Section 8 concludes this paper.

2 Background

In this section, we first overview the notion of payment channels and we then describe payment-channel networks.

2.1 Payment Channels

A payment channel enables several Bitcoin payments between two users without committing every single payment to the Bitcoin blockchain. The cornerstone of payment channels is depositing bitcoins into a multi-signature address controlled by both users and having the guarantee that all bitcoins are refunded at a mutually agreed time if the channel expires. In the following, we overview the basics of payment channels and we refer the reader to [60, 32, 50] for further details.

In the illustrative example depicted in Figure 1, Alice opens a payment channel with Bob with an initial capacity of 5 bitcoins. This *opening* transaction makes sure that Alice gets the money back after a certain timeout if the payment channel is not used. Now, Alice can pay off-chain to Bob by adjusting the balance of the deposit in favor of Bob. Each off-chain payment augments the balance for Bob and reduces it for Alice. When no more off-chain payments are needed (or the capacity of the payment channel is exhausted), the payment channel is closed with a *closing* transaction included in the blockchain. This transaction sends the deposited bitcoins to each user according the most recent balance in the payment channel.

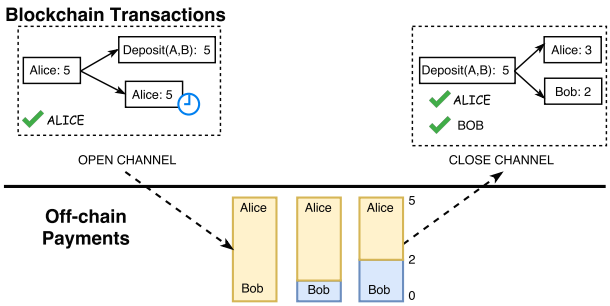

Figure 1: **Illustrative example of payment channel.** White solid boxes denote Bitcoin addresses and their current balance, dashed boxes represent Bitcoin transactions, the clock denotes a time lock contract [7], a user name along a tick denotes her signature to validate the transaction and colored boxes denote the state of the payment channel. Dashed arrows denote temporal sequence. Alice first deposits 5 bitcoins opening a payment channel with Bob, then uses it to pay Bob off-chain. Finally, the payment channel is closed with the most recent balance.

The payment channel depicted in Figure 1 is an example of *unidirectional* channel: it can be used only for payments from Alice to Bob. *Bidirectional* channels are defined to overcome this limitation as off-chain payments in both directions are possible. Bidirectional payment channels operate in essence as the unidirectional version.[1] The major technical challenge consists in changing the direction of the channel. In the running example, assume that the current payment channel balance *bal* is {Alice: 4, Bob: 1} and further assume that Bob pays off-chain one bitcoin back to Alice. The new payment channel balance *bal'* is {Alice: 5, Bob: 0}. At this point, Alice benefits from *bal'* balance while Bob benefits from *bal*. The solution to this discrepancy consists on that Bob and Alice make sure that any previous balance has been invalidated in favor of the most recent one. Different "invalidation" techniques have been proposed and we refer the reader to [60, 32, 65] for details.

The Bitcoin protocol has been updated recently to fully support payment channels. In particular, transaction malleability [8], along with a set of other interesting new features, have been added to the Bitcoin protocol with the recent adoption of Segregated Witness [16]. This event paves the way to the implementation and testing of PCNs on the main Bitcoin blockchain as of today [70].

[1]Technically, a bidirectional channel might require that both users contribute funds to the deposit in the opening transaction. However, current proposals [39] allow bidirectional channels with single deposit funder.

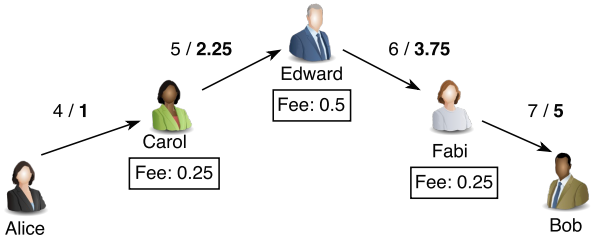

Figure 2: **Illustrative example of a payment in a PCN.** Non-bold (bold) numbers represent the capacity of the channels before (after) the payment from Alice to Bob. Alice wants to pay 2 bitcoins to Bob via Carol, Edward and Fabi. Therefore, she starts the payment with 3 bitcoins (i.e., payment amount plus fees).

2.2 A Payment Channel Network (PCN)

A PCN can be represented as a directed graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, where the set $\mathbb{V}$ of vertices represents the Bitcoin accounts and the set $\mathbb{E}$ of weighted edges represents the payment channels. Every vertex $u \in \mathbb{V}$ has associated a non-negative number that denotes the fee it charges for forwarding payments. The weight on a directed edge $(u_1, u_2) \in \mathbb{E}$ denotes the amount of remaining bitcoins that u_1 can pay to u_2. For ease of explanation, in the rest of the paper we represent a bidirectional channel between u_1 and u_2 as two directed edges, one in each direction.[2] Such a network can be used then to perform off-chain payments between two users that do not have an open channel between them but are connected by a path of open payment channels.

The success of a payment between two users depends on the capacity available along a path connecting the two users and the fees charged by the users in such path. Assume that s wants to pay α bitcoins to r and that they are connected through a path $s \to u_1 \to \ldots \to u_n \to r$. For their payment to be successful, every link must have a ca-

[2]In practice, there is a subtle difference: In a bidirectional channel between Alice and Bob, Bob can always return to Alice the bitcoins that she has already paid to him. However, if two unidirectional channels are used, Bob is limited to pay to Alice the capacity of the edge Bob $\to$ Alice, independently of the bitcoins that he has received from Alice. Nevertheless, our simplification greatly ease the understanding of the rest of the paper and proposed algorithms can be easily extended to support bidirectional channels.

pacity $\gamma_i \geq \alpha'_i$, where $\alpha'_i = \alpha - \sum_{j=1}^{i-1} fee(u_j)$ (i.e., the initial payment value minus the fees charged by intermediate users in the path). At the end of a successful payment, every edge in the path from s to r is decreased by α'_i. To ensure that r receives exactly α bitcoins, s must start the payment with a value $\alpha^* = \alpha + \sum_{j=1}^{n} fee(u_j)$.

In the illustrative example of payment shown in Figure 2, assume that Alice wants to pay Bob 2 bitcoins. For that she needs to start a payment for a value of 3 bitcoins (2 bitcoins plus 1 bitcoin for the fees charged by users in the path). Then the payment is settled as follows: capacity in the link Alice $\rightarrow$ Carol is reduced by 3. Additionally, Carol charges a fee if 0.25 bitcoins by reducing the capacity of the link Carol $\rightarrow$ Edward by 2.75 instead of 3 bitcoins. Following the same reasoning, the link Edward $\rightarrow$ Fabi is set to capacity 3.75 and the link Fabi $\rightarrow$ Bob is set to 5.

2.3 State-of-the-Art PCNs

The concepts of payment channels [41, 47, 32] and PCNs [50] have already attracted attention from the research community. In practice, there exist several ongoing implementations for a PCN in Bitcoin [11, 12, 10, 19]. Among them, the Lightning Network has emerged as the most prominent example in the Bitcoin community and an alpha implementation has been released recently [1]. The idea of a PCN has been proposed to improve scalability issues not only in Bitcoin, but also in other blockchain-based payment systems such as Ethereum [13].

2.3.1 Routing in PCNs

An important task in PCNs is to find paths with enough capacity between sender and receiver. In our setting, the network topology is known to every user. This is the case since the opening of each payment channel is logged in the publicly available blockchain. Additionally, a gossip protocol between users can be carried out to broadcast the existence of any payment channel [62]. Furthermore, the fees charged by every user can be made public by similar means. Under these conditions, the sender can locally calculate the paths between the sender and the receiver. In the rest of the paper, we assume that the sender chooses the path according to her own criteria. Nevertheless, we consider path selection as an interesting but orthogonal problem.

2.3.2 Payments in PCNs

A payment along a path of payment channels is carried out by updating the capacity of each payment channel in the path according to the payment amount and the associated fees (see Section 2.2). Such an operation rises the important challenge of *atomicity*: either the capacity of all channels in the path is updated or none of the channels is changed. Allowing changes in only some of the channels in the path can lead to the loss of bitcoins for a user (e.g., a user could pay certain bitcoins to the next user in the path but never receive the corresponding bitcoins from the previous neighbor).

The current proposal in the Lightning Network consists of a smart contract called *Hash Time-Lock Contract* (HTLC) [60]. This contract locks x bitcoins that can be released only if the contract is fulfilled. The contract is defined, in terms of a hash value $y := H(R)$ where R is chosen uniformly at random, the amount of bitcoins x and a timeout t, as follows:

> **HTLC (Alice, Bob, y, x, t):**
> 1. If Bob produces the condition R^* such that $H(R^*) = y$ before t days,[3] Alice pays Bob x bitcoins.
> 2. If t days elapse, Alice gets back x bitcoins.

An illustrative example of the use of HTLC in a payment is depicted in Figure 3. For simplicity, we

[3] We use days here as in the original description [60]. Instead, recent proposals use the sequence numbers of blocks as they appear in the Bitcoin blockchain [35].

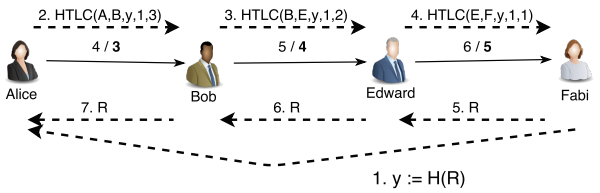

Figure 3: **Illustrative example of a payment from Alice to Fabi for value** 1 **using HTLC contract.** First, the condition is sent from Fabi to Alice. The condition is then forwarded among users in the path to hold 1 bitcoin at each payment channel. Finally, the receiver shows R, releasing the held bitcoin at each payment channel. For simplicity, we assume that there are no payment fees in this example.

assume that there are not payment fees in this example. First, the payment amount (i.e., 1 bitcoin) is set on hold from the sender to the receiver and then released from the receiver to the sender. In a bit more detail, after the receiver (Fabi) sends the condition to the sender (Alice), Alice sets an HTLC with her neighbor, effectively setting the payment value (i.e., 1 bitcoin) on hold. Such HTLC is then set at each payment channel in the path to the receiver. At this point, the receiver knows that the payment value is on hold at each payment channel and thus she reveals the value R, that allows her to fulfill the contract and to settle the new capacity at each payment channel in the path.

It is important to note that every user in the path sets the HTLC in the outgoing payment channel with a timeout smaller than the HTLC in the incoming payment channel. In this manner, the user makes sure that she can pull bitcoins from her predecessor after her bitcoins have been pulled from her successor. An offline user can outsource the monitoring of fulfillments corresponding to open HTLC contracts associated to her payment channels [33].

Although HTLC is fully compatible with Bitcoin, its use in practice leads to important privacy leaks: It is easy to see that the value of the hash $H(R)$ uniquely identifies the users that took part in a specific transaction. This fact has two main implications. First, any two colluding users in a path can trivially derive the fact that they took part in the same payment and this can be leveraged to reconstruct the identity of sender and receiver.[4] Second, if the HTLC statements are uploaded to the blockchain (e.g., due to uncollaborative intermediate users in the payment path), an observer can easily track the complete path used to route the payment, even if she is not part of the payment. In this work, we propose a novel Multi-Hop HTLC smart contract that avoids this privacy problem while ensuring that no intermediate user loses her bitcoins.

An important issue largely understudied in current PCNs is the handling of concurrent payments that require a *shared* payment channel in their paths. Current proposals simply abort a payment if the balance at the shared payment channel in the path is not enough. However, as we show in Section 3.3, this approach can lead to a deadlock situation where none of simultaneous payments terminates. We propose a payment protocol that ensure non-blocking progress, that is, at least one of the concurrent payments terminates. Moreover, we show an inherent tradeoff between concurrency and privacy for any fully distributed payment network.

[4] As noted in [40], in a path $A \to I_1 \to I_2 \to I_3 \to B$, only I_1 and I_3 must collude to recover the identities of A and B as all the contracts in the path share the same $H(R)$.

3 Problem Definition

In this section, we first formalize a PCN and underlying operations, and discuss the attacker model and our security and privacy goals. We then describe an ideal world functionality for our proposal, and present a system overview. Throughout the following description we implicitly assume that every algorithm takes as input the blockchain, which is publicly known to all users.

Definition 1 (Payment Channel Network (PCN))**.** *A PCN is defined as graph $\mathbb{G} := (\mathbb{V}, \mathbb{E})$, where $\mathbb{V}$ is the set of Bitcoin accounts and $\mathbb{E}$ is the set of currently open payment channels. A PCN is defined with respect to a blockchain* B *and is equipped with the three operations (*$\mathsf{openChannel}$, $\mathsf{closeChannel}$, pay*) described below:*

- $\mathsf{openChannel}(u_1, u_2, \beta, t, f) \to \{1, 0\}$. *On input two Bitcoin addresses $u_1, u_2 \in \mathbb{V}$, an initial channel capacity β, a timeout t, and a fee value f, if the operation is authorized by u_1, and u_1 owns at least β bitcoins,* $\mathsf{openChannel}$ *creates a new payment channel $(c_{\langle u_1,u_2\rangle}, \beta, f, t) \in \mathbb{E}$, where $c_{\langle u_1,u_2\rangle}$ is a fresh channel identifier. Then it uploads it to* B *and returns* 1*. Otherwise, it returns* 0*.*
- $\mathsf{closeChannel}(c_{\langle u_1,u_2\rangle}, v) \to \{1, 0\}$. *On input a channel identifier $c_{\langle u_1,u_2\rangle}$ and a balance v (i.e., the distribution of bitcoins locked in the channel between u_1 and u_2), if the operation is authorized by both u_1 and u_2,* $\mathsf{closeChannel}$ *removes the corresponding channel from $\mathbb{G}$, includes the balance v in* B *and returns* 1*. Otherwise, it returns* 0*.*
- $\mathsf{pay}((c_{\langle s,u_1\rangle}, \ldots, c_{\langle u_n,r\rangle}), v) \to \{1, 0\}$. *On input a list of channel identifiers $(c_{\langle s,u_1\rangle}, \ldots, c_{\langle u_n,r\rangle})$ and a payment value v, if the payment channels form a path from the sender (s) to the receiver (r) and each payment channel $c_{\langle u_i,u_{i+1}\rangle}$ in the path has at least a current balance $\gamma_i \geq v'_i$, where $v'_i = v - \sum_{j=1}^{i-1} fee(u_j)$, the* pay *operation decreases the current balance for each payment channel $c_{\langle u_i,u_{i+1}\rangle}$ by v'_i and returns* 1*. Otherwise, none of the balances*

at the payment channels is modified and the pay *operation returns* 0.

3.1 Attacker Model, and Security and Privacy Goals

We consider a computationally efficient attacker that can shape the network at her will by spawning users and corrupting an arbitrary subset of them in an adaptive fashion. Once a user is corrupted, its internal state is given to the attacker and all of the following messages for that user are handed over to the attacker. On the other hand, we assume that the communication between two non-compromised users sharing a payment channel is confidential (e.g., through TLS). Finally, the attacker can send arbitrary messages on behalf of corrupted users.

Against the above adversary, we identify the following security and privacy notions of interest:

- **Balance security.** Intuitively, balance security guarantees that any honest intermediate user taking part in a pay operation (as specified in Definition 1) does not lose coins even when all other users involved in the pay operation are corrupted.
- **Serializability.** We require that the executions of PCN are *serializable* [58], i.e., for every concurrent execution of pay operations, there exists an *equivalent* sequential execution.
- **(Off-path) Value Privacy.** Intuitively, value privacy guarantees that for a pay operation involving only honest users, corrupted users outside the payment path learn no information about the payment value.
- **(On-path) Relationship Anonymity [59, 24].** Relationship anonymity requires that, given two simultaneous successful pay operations of the form $\left\{\mathsf{pay}_i((c_{\langle s_i,u_1\rangle},\ldots,c_{\langle u_n,r_i\rangle}),v)\right\}_{i\in[0,1]}$ with at least one honest intermediate user $u_{j\in[1,n]}$, corrupted intermediate users cannot determine the pair (s_i,r_i) for a given pay_i with probability better than $1/2$.

3.2 Ideal World Functionality

Our Model. The users of the network are modeled as interactive Turing machines that communicate with a trusted functionality $\mathcal{F}$ via secure and authenticated channels. We model the attacker $\mathcal{A}$ as a probabilistic polynomial-time machine that is given additional interfaces to add users to the system and corrupt them. $\mathcal{A}$ can query those interfaces adaptively and at any time. Upon corruption of a user u, the attacker is provided with the internal state of u and the incoming and outgoing communication of u is routed thorough $\mathcal{A}$.

Assumptions. We model anonymous communication between any two users of the network as an ideal functionality $\mathcal{F}_{\mathsf{anon}}$, as proposed in [26]. Furthermore, we assume the existence of a blockchain B that we model as a trusted append-only bulletin board (such as [72]): The corresponding ideal functionality $\mathcal{F}_{\mathsf{B}}$ maintains B locally and updates it according to the transactions between users. At any point in the execution, any user u of the PCN can send a distinguished message read to $\mathcal{F}_{\mathsf{B}}$, who sends the whole transcript of B to u. We denote the number of entries of B by $|\mathsf{B}|$. In our model, time corresponds to the number of entries of the blockchain B, i.e., time t is whenever $|\mathsf{B}| = t$. Our idealized process $\mathcal{F}$ uses $\mathcal{F}_{\mathsf{anon}}$ and $\mathcal{F}_{\mathsf{B}}$ as subroutines, i.e., our protocol is specified in the $(\mathcal{F}_{\mathsf{anon}},\mathcal{F}_{\mathsf{B}})$-hybrid model. Note that our model for a blockchain is a coarse grained abstraction of the reality and that more accurate formalizations are known in the literature, see [44]. For ease of exposition we stick to this simplistic view, but one can easily extend our model to incorporate more sophisticated abstractions.

Notation. Payment channels in the Blockchain B are of the form $(c_{\langle u,u'\rangle},v,t,f)$, where $c_{\langle u,u'\rangle}$ is a unique channel identifier, v is the capacity of the channel, t is the expiration time of the channel, and f is the associated fee. For ease of notation we assume that the identifiers of the users (u,u') are also encoded in $c_{\langle u,u'\rangle}$. We stress that any two users may have multiple channels open simultaneously. The functionality maintains two additional internal lists $\mathcal{C}$ and $\mathcal{L}$. The former is used to keep track of the closed channels, while the latter records the off-chain payments. Entries in $\mathcal{L}$ are of the form $(c_{\langle u,u'\rangle},v,t,h)$, where $c_{\langle u,u'\rangle}$ is the corresponding channel, v is the amount of credit used, t is the expiration time of the payment, and h is the identifier for this entry.

Operations. In Figure 4 we describe the interactions between $\mathcal{F}$ and the users of the PCN. For simplicity, we only model unidirectional channels, although our functionality can be easily extended to

Open channel: On input $(\mathsf{open}, c_{\langle u,u'\rangle}, v, u', t, f)$ from a user u, the $\mathcal{F}$ checks whether $c_{\langle u,u'\rangle}$ is well-formed (contains valid identifiers and it is not a duplicate) and eventually sends $(c_{\langle u,u'\rangle}, v, t, f)$ to u', who can either abort or authorize the operation. In the latter case, $\mathcal{F}$ appends the tuple $(c_{\langle u,u'\rangle}, v, t, f)$ to B and the tuple $(c_{\langle u,u'\rangle}, v, t, h)$ to $\mathcal{L}$, for some random h. $\mathcal{F}$ returns h to u and u'.

Close channel: On input $(\mathsf{close}, c_{\langle u,u'\rangle}, h)$ from a user $\in \{u', u\}$ the ideal functionality $\mathcal{F}$ parses B for an entry $(c_{\langle u,u'\rangle}, v, t, f)$ and $\mathcal{L}$ for an entry $(c_{\langle u,u'\rangle}, v', t', h)$, for $h \neq \perp$. If $c_{\langle u,u'\rangle} \in \mathcal{C}$ or $t > |\mathsf{B}|$ or $t' > |\mathsf{B}|$ the functionality aborts. Otherwise, $\mathcal{F}$ adds the entry $(c_{\langle u,u'\rangle}, u', v', t')$ to B and adds $c_{\langle u,u'\rangle}$ to $\mathcal{C}$. $\mathcal{F}$ then notifies both users involved with a message $(c_{\langle u,u'\rangle}, \perp, h)$.

Payment: On input $(\mathsf{pay}, v, (c_{\langle u_0,u_1\rangle}, \ldots, c_{\langle u_n,u_{n+1}\rangle}), (t_0, \ldots, t_n))$ from a user u_0, $\mathcal{F}$ executes the following interactive protocol:

1. For all $i \in \{1, \ldots, (n+1)\}$ $\mathcal{F}$ samples a random h_i and parses B for an entry of the form $(c_{\langle u_{i-1},u'_i\rangle}, v_i, t'_i, f_i)$. If such an entry does exist $\mathcal{F}$ sends the tuple $(h_i, h_{i+1}, c_{\langle u_{i-1},u_i\rangle}, c_{\langle u_i,u_{i+1}\rangle}, v - \sum_{j=i}^{n} f_j, t_{i-1}, t_i)$ to the user u_i via an anonymous channel (for the specific case of the receiver the tuple is only $(h_{n+1}, c_{\langle u_n,u_{n+1}\rangle}, v, t_n)$). Then $\mathcal{F}$ checks whether for all entries of the form $(c_{\langle u_{i-1},u_i\rangle}, v'_i, \cdot, \cdot) \in \mathcal{L}$ it holds that $v'_i \geq \left(v - \sum_{j=i}^{n} f_j\right)$ and that $t_{i-1} \geq t_i$. If this is the case $\mathcal{F}$ adds $d_i = (c_{\langle u_{i-1},u_i\rangle}, (v'_i - (v - \sum_{j=i}^{n} f_j)), t_i, \perp)$ to $\mathcal{L}$, where $(c_{\langle u_{i-1},u_i\rangle}, v'_i, \cdot, \cdot) \in \mathcal{L}$ is the entry with the lowest v'_i. If any of the conditions above is not met, $\mathcal{F}$ removes from $\mathcal{L}$ all the entries d_i added in this phase and aborts.

2. For all $i \in \{(n+1), \ldots, 1\}$ $\mathcal{F}$ queries all u_i with (h_i, h_{i+1}), through an anonymous channel. Each user can reply with either $\top$ or $\perp$. Let j be the index of the user that returns $\perp$ such that for all $i > j : u_i$ returned $\top$. If no user returned $\perp$ we set $j = 0$.

3. For all $i \in \{j+1, \ldots, n\}$ the ideal functionality $\mathcal{F}$ updates $d_i \in \mathcal{L}$ (defined as above) to $(-, -, -, h_i)$ and notifies the user of the success of the operation with with some distinguished message $(\mathsf{success}, h_i, h_{i+1})$. For all $i \in \{0, \ldots, j\}$ (if $j \neq 0$) $\mathcal{F}$ removes d_i from $\mathcal{L}$ and notifies the user with the message $(\perp, h_i, h_{i+1})$.

Figure 4: Ideal world functionality for PCNs.

support also bidirectional channels. The execution of our simulation starts with $\mathcal{F}$ initializing a pair of local empty lists $(\mathcal{L}, \mathcal{C})$. Users of a PCN can query $\mathcal{F}$ to open channels and close them to any valid state in $\mathcal{L}$. On input a value v and a set of payment channels $(c_{\langle u_0,u_1\rangle}, \ldots, c_{\langle u_n,u_{n+1}\rangle})$ from some user u_0, $\mathcal{F}$ checks whether the path has enough capacity (step 1) and initiates the payment. Each intermediate user can either allow the payment or deny it. Once the payment has reached the receiver, each user can again decide to interrupt the flow of the payment (step 2), i.e., pay instead of the sender. Finally $\mathcal{F}$ informs the involved nodes of the success of the operation (step 3) and adds the updated state to $\mathcal{L}$ for the corresponding channels.

Discussion. Here, we show that our ideal functionality captures the security and privacy properties of interest for a PCN.

• *Balance security.* Let u_i be any intermediate hop in a payment $\mathsf{pay}((c_{\langle s,u_1\rangle}, \ldots, c_{\langle u_n,r\rangle}), v)$. $\mathcal{F}$ locally updates in $\mathcal{L}$ the channels corresponding to the incoming and outgoing edges of u_i such that the total balance of u_i is increased by the coins she sets as a fee, unless the user actively prevents it (step 2). Since $\mathcal{F}$ is trusted, balance security follows.

• *Serializability.* Consider for the moment only single-hop payments. It is easy to see that the ideal functionality executes them serially, i.e., any two concurrent payments can only happen on different links. Therefore one can trivially find a scheduler that performs the same operation in a serial order (i.e., in any order). By balance security, any payment can be represented as a set of atomic single-hop payments and thus serializability holds.

• *Value Privacy.* In the ideal world, users that do not lie in the payment path are not contacted by $\mathcal{F}$ and therefore they learn nothing about the transacted value (for the off-chain payments).

• *Relationship Anonymity.* Let u_i be an intermediate hop in a payment. In the interaction with the ideal functionality, u_i is only provided with a unique identifier for each payment. In particular, such an identifier is completely independent from the identifiers of other users involved in the same payment. It follows that, as long as at least one honest user u_i lies in a payment path, any two simultaneous payments over the same path for the same value v are indistinguishable to the eyes of the user u_{i+1}. This implies that any *proper* subset of corrupted intermediate hops, for any two successful concurrent payments traversing all of the corrupted nodes, cannot distinguish in which order an honest u_i forwarded the payments. Therefore such a set of corrupted nodes cannot determine the correct sender-receiver pair with probability better than 1/2.

UC-Security. Let $\mathsf{EXEC}_{\tau,\mathcal{A},\mathcal{E}}$ be the ensemble of the outputs of the environment $\mathcal{E}$ when interacting with the adversary $\mathcal{A}$ and parties running the protocol τ (over the random coins of all the involved machines).

Definition 2 (UC-Security)**.** *A protocol τ UC-realizes an ideal functionality $\mathcal{F}$ if for any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that for any environment $\mathcal{E}$ the ensembles $\mathsf{EXEC}_{\tau,\mathcal{A},\mathcal{E}}$ and $\mathsf{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{E}}$ are computationally indistinguishable.*

Lower bound on byzantine users in PCN. We observe that in PCNs that contain channels in which both the users are *byzantine* (à la malicious) [46], there is an inherent cost to concurrency. Specifically, in such a PCN, if we are providing non-blocking progress, i.e., at least one of the concurrent payments terminates, then it is impossible to provide serializability in PCNs (cf. Figure 11 in Appendix D). Thus, henceforth, all results and claims in this paper assume that in any PCN execution, there does not exist a channel in which both its users are byzantine.

Lemma 1. *There does not exist any serializable protocol for the PCN problem that provides non-blocking progress if there exists a payment channel in which both users are byzantine.*

3.3 Key Ideas and System Overview

In the following, we give a high-level overview on how we achieve private and concurrent payments in PCNs.

3.3.1 Payment Privacy

The payment operation must ensure the security and privacy properties of interest in a PCN, namely balance security, value privacy and relationship anonymity. A naïve approach towards achieving balance security would be to use HTLC-based payments (see Section 2.3.2). This solution is however in inherent conflict with anonymity: It is easy to see that contracts belonging to the same transactions are *linkable* among each other, since they encode the same condition (h) to release the payment. Our proposal, called Multi-Hop HTLC, aims to remove this link among hops while maintaining the full compatibility with the Bitcoin network.

The idea underlying Multi-Hop HTLC is the following: At the beginning of an n-hop transaction the sender samples n-many independent strings $(x_1, \ldots, x_n)$. Then, for all $i \in 1, \ldots, n$, she sets $y_i = H\left(\bigoplus_{j=i}^{n} x_j\right)$, where H is an arbitrary hash function. That is, each y_i is the result of applying the function H to all of the input values x_j for $j \geq i$ in an XOR combiner. The sender then provides the receiver with (y_n, x_n) and the i-th node with the tuple (y_{i+1}, y_i, x_i). In order to preserve anonymity, the sender communicates those values to the intermediate nodes over an anonymous channel. Starting from the *sender*, each pair of neighboring nodes (u_{i+1}, u_i) defines a standard HTLC on inputs $(u_i, u_{i+1}, y_i, b, t)$, where b and t are the amount of bitcoin and the timeout parameter, respectively. Note that the release conditions of the contracts are uniformly distributed in the range of the function H and therefore the HTLCs of a single transaction are independent from each other. Clearly, the mechanism described above works fine as long as the sender chooses each value y_i according to the specification of the protocol. We can enforce an honest behavior by including non-interactive zero-knowledge proofs [38].

3.3.2 Concurrent Payments

It is possible that two (or more) simultaneous payments share a payment channel in their payment

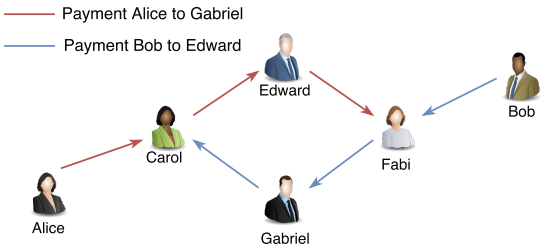

Figure 5: **Illustrative example of two blocking payments:** Alice to Gabriel (red) and Bob to Edward (blue). For simplicity, assume each payment pays 1 bitcoin and each payment channel has capacity 1 bitcoin. Each payment channel is colored with the payment that has reached it first. In this deadlock situation, none of the payments can continue further in the path and cannot be trivially completed.

paths in such a manner that none of the payments goes through. In the example depicted in Figure 5, the payment from Alice to Gabriel cannot be carried out as the capacity in the payment channel between Fabi and Gabriel is already locked for the payment from Bob to Edward. Moreover, this second payment cannot be carried out either as the capacity on the payment channel between Carol and Edward is already locked. This deadlock situation is a generic problem of PCNs, where a payment is aborted as soon as there exists a payment channel in the path without enough capacity.

Blocking Payments (Fulgor). A best-effort solution for avoiding this *deadlock* consists on letting both payments fail. Aborted payments do not affect the balance of the involved users as the receiver would not disclose the release condition for the locked payment channels. Therefore, involved payment channels would get unlocked only after the corresponding timeout and bitcoins are sent back to the original owner.

The sender of an aborted payment can then randomly choose a waiting period to reissue the payment. Although the *blocking* mechanism closely resembles the practice of users in others payment networks such as Ripple [14] or SilentWhispers [49], it might degrade transaction throughput in a fully decentralized PCN.

Non-blocking Payments (Rayo). An alternative solution consists on a *non-blocking* solution where at least one out of a set of concurrent payments completes. Our approach to achieve it assumes that there exists a global ordering of payments (e.g., by a global payment identifier). In a nutshell, users can queue payments with higher identifier than the current one "in-flight", and abort payments with lower identifiers. This ensures that either the current in-flight payment completes or one of the queued payments would do, as their identifiers are higher.

4 Fulgor: Our Construction

In this section, we introduce the cryptographic building blocks required for our construction (Section 4.1), we describe the details for the Multi-Hop HTLC contract (Section 4.2), we detail the constructions for PCN operations (Section 4.3), analyze its security and privacy (Section 4.4) and conclude with a few remarks (Section 4.5).

Notation. We denote by λ the security parameter of our system and we use the standard definition for a *negligible* function. We denote by *decision* the possible events in a payment channel due to a payment. The *decision* forward signals to lock the balance in the payment channel corresponding to the payment value. The *decision* abort signals the release of locked funds in the payment channel due to the abortion of a payment. Correspondingly, the *decision* accept signals the confirmation of a payment accepted by the receiver.

For ease of notation, we assume that users identifiers (u_i, u_{i+1}) can be extracted from the channel identifier $c_{\langle u_i, u_{i+1}\rangle}$.

System Assumptions. We assume that every user in the PCN is aware of the complete network topology, that is, the set of all users and the existence of a payment channel between every pair of users. We further assume that the sender of a payment chooses a payment path 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 as otherwise privacy is trivially broken. We further assume that every user is aware of the payment fees charged by each other user in the PCN.

This can be accomplished in practice. The opening of a payment channel between two users requires to add a transaction in the blockchain that includes both user identifiers. Therefore, the topology of the PCN is trivially leaked. Moreover, the transaction used to open a payment channel can contain user-

defined data [5] so that each user can embed her own payment fee. In this manner, each user can proactively gather updated information about the network topology and fees from the blockchain itself or be disseminated by a gossip protocol [62, 48].

We further assume that pairs of users sharing a payment channel communicate through secure and authenticated channels (such as TLS), which is easy to implement given that every user is uniquely identified by a public key. Also we assume that the sender and the receiver of a (possibly indirect) transaction can communicate through a secure and direct channel. Finally, we assume that the sender of a payment can create an anonymous payment channel with each intermediate user. The IP address where to reach each user could be encoded in the channel creation transaction and therefore logged in the blockchain. We note that our protocol is completely parametric with respect to the routing, therefore any onion routing-like techniques would work in this context.

We consider the *bounded synchronous* communication setting [23]. In such communication model, time is divided into fixed communication rounds and it is assumed that all messages sent by a user in a round are available to the intended recipient within a bounded number of steps in an execution. Consequently, absence of a message indicates absence of communication from a user during the round. In practice, this can be achieved with loosely synchronized clocks among the users in the PCN [28].

Finally, we assume that there is a total order among the users (e.g., lexicographically sorted by their public verification keys).

4.1 Building Blocks

Non-Interactive Zero-Knowledge. Let $\mathcal{R} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$ be an NP relation, and let $\mathcal{L}$ be the set of positive instances for $\mathcal{R}$, i.e., $\mathcal{L} = \{x \mid \exists w \text{ s.t. } \mathcal{R}(x,w) = 1\}$. A non-interactive zero-knowledge proof for $\mathcal{R}$ consists of a single message from a prover $\mathcal{P}$ to a verifier $\mathcal{V}$. The prover $\mathcal{P}$ wants to compute a proof π that convinces the verifier $\mathcal{V}$ that a certain statement $x \in \mathcal{L}$. We allow the prover to run on an extra private input w such that $\mathcal{R}(x,w) = 1$. The verifier can either accept or reject, depending on π. A NIZK is complete if the $\mathcal{V}$ always accepts honestly computed π for a statement $x \in \mathcal{L}$ and it is sound if $\mathcal{V}$ always rejects any π for all $x \notin \mathcal{L}$, except with negligible probability. Loosely speaking, a NIZK proof is zero knowledge if the verifier learns nothing from π beyond the fact that $x \in \mathcal{L}$. Efficient NIZK protocols are known to exist in the random oracle model [38].

Two Users Agreement. Two users u_i and u_j sharing a payment channel, locally maintain the state of the payment channel defined as a scalar *channel-state* $:= \mathsf{cap}(c_{\langle u_i,u_j\rangle})$ that denotes the current capacity of their payment channel. We require a two party agreement protocol that ensures that both users agree on the current value of $\mathsf{cap}(c_{\langle u_i,u_j\rangle})$ at each point in time. We describe the details of such protocol in Appendix B. For readability, in the rest we implicitly assume that two users sharing a payment channel satisfactorily agree on its current state.

4.2 Multi-Hop HTLC

We consider the standard scenario of an indirect payment from a sender Sdr to a receiver Rvr for a certain value v through a path of users $(u_1, \ldots, u_n)$, where $u_n = \mathsf{Rvr}$. All users belonging to the same network share the description of a hash function $H : \{0,1\}^* \to \{0,1\}^\lambda$ that we model as a random oracle.

Let $\mathcal{L}$ be the following language: $\mathcal{L} = \{(H, y', y, x) \mid \exists(w) \text{ s.t. } y' = H(w) \wedge y = H(w \oplus x)\}$ where $w \oplus x$ denotes the bitwise XOR of the two bitstrings. Before the payment starts, the sender Sdr locally executes the following $\mathsf{Setup}_{\mathrm{HTLC}}$ algorithm described in Figure 6.

$\mathsf{Setup}_{\mathrm{HTLC}}(n)$:

$\forall i \in [n]$:

$$x_i \in \{0,1\}^\lambda; y_i \leftarrow H\left(\bigoplus_{j=i}^{n} x_j\right)$$

$\forall i \in [n-1]$:

$$\pi_i \leftarrow \mathcal{P}\left((H, y_{i+1}, y_i, x_i), \left(\bigoplus_{j=i+1}^{n} x_j\right)\right)$$

return $((x_1, y_1, \pi_1), \ldots, (x_n, y_n))$

Figure 6: Setup operation for the Multi-Hop HTLC contract.

Intuitively, the sender samples n-many random strings x_i and defines y_i as $H\left(\bigoplus_{j=i}^{n} x_j\right)$ which is the XOR combination of all x_j such that $j \geq i$. Then, Sdr computes the proofs π to guarantee that each y_i is well-formed, without revealing all of the x_i. The receiver is provided with (x_n, y_n) and she simply checks that $y_n = H(x_n)$. Sdr then sends (x_i, y_i, π_i) to each intermediate user u_i, through a direct communication channel. Each u_i runs $\mathcal{V}((H, y_{i+1}, y_i, x_i), \pi_i)$ and aborts the payment if the verification algorithm rejects the proof.

Starting from the user $u_0 = \mathsf{Sdr}$, each pair of users (u_i, u_{i+1}) check whether both users received the same values of (y_{i+1}, v). This can be done by simply exchanging and comparing the two values. If this is the case, they establish HTLC $(u_i,\ u_{i+1},\ y_{i+1},\ v,\ t_i)$ as described in Section 2.3, where t_i defines some timespan such that for all $i \in [n]$: $t_{i-1} = t_i + \Delta$, for some positive value Δ. Once the contract between (u_{n-1}, u_n) is settled, the user u_n (the receiver) can then pull v bitcoins by releasing the x_n, which by definition satisfies the constraint $H(x_n) = y_n$. Once the value of x_n is published, u_{n-1} can also release a valid condition for the contract between (u_{n-2}, u_{n-1}) by simply outputting $x_{n-1} \oplus x_n$. In fact, this mechanism propagates for every intermediate user of the payment path, until Sdr: For each node u_i it holds that, whenever the condition for the contract between (u_i, u_{i+1}) is released, i.e., somebody publishes a string r such that $H(r) = y_{i+1}$, then u_i immediately learns $x_i \oplus r$ such that $H(x_i \oplus r) = y_i$, which is a valid condition for the contract between (u_{i-1}, u_i). It follows that each intermediate user whose *outgoing* contract has been pulled is able to release a valid condition for the *incoming* contract.

4.3 Construction Details

In the following, we describe the details of the three operations (openChannel, closeChannel, pay) that compose Fulgor.

- openChannel(u_1, u_2, β, t, f): The purpose of this operation is to open a payment channel between users u_1 and u_2. For that, they create an initial Bitcoin deposit that includes the following information: their Bitcoin addresses, the initial capacity of the channel (β), the channel timeout (t), the fee charged to use the channel (f) and a channel identifier $(c_{\langle u_1,u_2\rangle})$ agreed beforehand between both users. After the Bitcoin deposit has been successfully added to the blockchain, the operation returns 1. If any of the previous steps is not carried out as defined, the operation returns 0.
- closeChannel$(c_{\langle u_1,u_2\rangle}, v)$: This operation is used by two users (u_1, u_2) sharing an open payment channel $(c_{\langle u_1,u_2\rangle})$ to close it at the state defined by v and accordingly update their bitcoin balances in the Bitcoin blockchain. This operation in Fulgor is performed as defined in the original proposal of payment channels (see Section 2.1), additionally returning 1 if and only if the corresponding Bitcoin transaction is added to the Bitcoin blockchain.
- pay$((c_{\langle u_0,u_1\rangle}, \ldots, c_{\langle u_n,u_{n+1}\rangle}), v)$: A payment operation transfers a value v from a sender (u_0) to a receiver (u_{n+1}) through a path of open payment channels between them $(c_{\langle u_0,u_1\rangle},\ \ldots,\ c_{\langle u_n,u_{n+1}\rangle})$. Here, we describe a *blocking* version of the payment operation (see Section 3.3). We discuss the non-blocking version of the payment operation in Section 5.

As shown in Figure 7 (black pseudocode), the

$\mathsf{pay}_{u_0}(m)$:

$(\mathsf{Txid}, \{c_{\langle u_0,u_1\rangle}\} \cup \{c_{\langle u_i,u_{i+1}\rangle}\}_{i\in[n]}, v) \leftarrow m$

$v_1 := v + \sum_{i}^{n} \mathrm{fee}(u_i)$

if $v_1 \leq \mathsf{cap}(c_{\langle u_0,u_1\rangle})$ **then**

  $\mathsf{cap}(c_{\langle u_0,u_1\rangle}) := \mathsf{cap}(c_{\langle u_0,u_1\rangle}) - v_1$

  $t_0 := t_{\text{now}} + \Delta \cdot n$

  $\forall i \in [n]$:

    $v_i := v_1 - \sum_{j=1}^{i-1} \mathrm{fee}(u_j)$

    $t_i := t_{i-1} - \Delta$

    $\{(x_i, y_i, \pi_i)\}_{i\in[n+1]} \leftarrow \mathsf{Setup}_{\mathrm{HTLC}}(n+1)$

    $\mathsf{Send}(u_i, ((\mathsf{Txid}, x_i, y_i, y_{i+1},$

      $\pi_i, c_{\langle u_{i-1},u_i\rangle}, c_{\langle u_i,u_{i+1}\rangle}, v_{i+1}, t_i, t_{i+1}), \mathsf{forward}))$

  $\mathbf{HTLC}(u_0, u_1, y_1, v_1, t_1)$

  $\mathsf{Send}(u_{n+1}, (\mathsf{Txid}, x_{n+1}, y_{n+1}, c_{\langle u_n,u_{n+1}\rangle},$

    $v_{n+1}, t_{n+1}))$

else

  abort

Figure 7: The pay routine in Fulgor for the sender. The light blue pseudocode shows additional steps required in Rayo.

$\mathsf{pay}_{u_{n+1}}(m)$:

$(\mathsf{Txid}, x_{n+1}, y_{n+1}, c_{\langle n,n+1\rangle}, v_{n+1}, t_{n+1}) \leftarrow m$
if $H(x_{n+1}) = y_{n+1}$ **and** $t_{n+1} > t_{\text{now}} + \Delta$ **then**
  store $(x_{n+1}, y_{n+1}, c_{\langle n,n+1\rangle}, t_{n+1})$
  $\mathsf{Send}(u_n, ((\mathsf{Txid}, x_{n+1}, y_{n+1}, c_{\langle n,n+1\rangle}), \mathsf{accept}))$
else
  $\mathsf{Send}(u_n, ((\mathsf{Txid}, y_{n+1}, c_{\langle n,n+1\rangle}, v_{n+1}), \mathsf{abort}))$

$\mathsf{pay}_{u_i}(m)$:

$(m^*, \mathit{decision}) \leftarrow m$
if $\mathit{decision} = \mathsf{forward}$ **then**
  $(\mathsf{Txid}, x_i, y_i, y_{i+1}, \pi_i, c_{\langle i-1,i\rangle}, c_{\langle i,i+1\rangle},$
    $v_{i+1}, t_i, t_{i+1}) \leftarrow m^*$
  if $v_{i+1} \leq \mathsf{cap}(c_{\langle u_i,u_{i+1}\rangle})$ **and** $\mathcal{V}((H, y_{i+1}, y_i, x_i), \pi_i)$
    and $t_{i+1} = t_i - \Delta$ **then**
    $\mathsf{cap}(c_{\langle u_i,u_{i+1}\rangle}) := \mathsf{cap}(c_{\langle u_i,u_{i+1}\rangle}) - v_{i+1}$
    HTLC$(u_i, u_{i+1}, y_{i+1}, v_{i+1}, t_{i+1})$
    $\mathit{cur}(c_{\langle u_i,u_{i+1}\rangle}).\mathit{append}(m^*)$
  else if $\exists k \mid \mathsf{Txid} > \mathit{cur}(c_{\langle u_i,u_{i+1}\rangle})[k].\mathsf{Txid}$ **then**
    $Q(c_{\langle u_i,u_{i+1}\rangle}).\mathit{append}(m^*)$
  else
    $\mathsf{Send}(u_{i-1}, ((\mathsf{Txid}, y_i, c_{\langle i-1,i\rangle}, v_i), \mathsf{abort}))$
else if $\mathit{decision} = \mathsf{abort}$ **then**
  $(\mathsf{Txid}, y_{i+1}, c_{\langle i,i+1\rangle}, v_{i+1}) \leftarrow m^*$
  $\mathsf{cap}(c_{\langle u_i,u_{i+1}\rangle}) := \mathsf{cap}(c_{\langle u_i,u_{i+1}\rangle}) + v_{i+1}$
  $\mathsf{Send}(u_{i-1}, ((\mathsf{Txid}, y_i, c_{\langle i-1,i\rangle}, v_i), \mathsf{abort}))$
  $\mathit{cur}(c_{\langle u_i,u_{i+1}\rangle}).\mathit{delete}(m^*.\mathsf{Txid})$
  $m' := \max(Q(c_{\langle u_i,u_{i+1}\rangle}))$
  $\mathsf{pay}_{u_i}((m', \mathsf{forward}))$
else if $\mathit{decision} = \mathsf{accept}$ **then**
  $(\mathsf{Txid}, x_{i+1}, y_{i+1}, c_{\langle i,i+1\rangle}, v_{i+1}) \leftarrow m^*$
  store $(x_{i+1} \oplus x_i, y_i, c_{\langle i-1,i\rangle}, t_i)$
  $\mathsf{Send}(u_{i-1}, ((\mathsf{Txid}, x_{i+1} \oplus x_i, y_i, c_{\langle i-1,i\rangle}, v_i), \mathsf{accept}))$
  $\mathit{cur}(c_{\langle u_i,u_{i+1}\rangle}).\mathit{delete}(m^*.\mathsf{Txid})$

Figure 8: The pay routine in Fulgor for the receiver and each intermediate user. The light blue pseudocode shows additional steps in Rayo. $\max(Q)$ returns the information for the payment with highest identifier among those in Q.

sender first calculates the cost of sending v bitcoins to Rvr as $v_1 := v + \sum_i \mathit{fee}(u_i)$, and the corresponding cost at each of the intermediate hops in the payment path. If the sender does not have enough bitcoins, she aborts the payment. Otherwise, the sender sets up the contract for each intermediate payment channel following the mechanism described in Section 4.2 and sends the information to the corresponding users.

Every intermediate user verifies that the incoming HTLC has an associated value smaller or equal than the capacity of the payment channel with her sucessor in the path. Additionally, every intermediate user verifies that the zero-knowledge proof associated to the HTLC for incoming and outgoing payment channels correctly verifies and that the timeout for the incoming HTLC is bigger than the timeout for the outgoing HTLC by a difference of Δ. If so, she generates the corresponding HTLC for the same associate value (possibly minus the fees) with the successor user in the path; otherwise, she aborts by triggering the abort event to the predecessor user in the path. These operations have been shown in Figure 8 (black pseudocode).

If every user in the path accepts the payment, it eventually reaches the receiver who in turn releases the information required to fulfill the HTLC contracts in the path (see Figure 8 (black pseudocode)). Interestingly, if any intermediate user aborts the payment, the receiver does not release the condition as she does not receive any payment. Moreover, payment channels already set in the previous hops of the path are voided after the timeout set in the corresponding HTLC.

4.4 Security and Privacy Analysis

In the following, we state the security and privacy results for Fulgor. We prove our results in the $(\mathcal{F}_{\mathsf{anon}}, \mathcal{F}_{\mathsf{B}})$-hybrid model. In other words, Theorem 1 holds for any UC-secure realization of $\mathcal{F}_{\mathsf{anon}}$ and $\mathcal{F}_{\mathsf{B}}$. We show the proof of Theorem 1 in Appendix A.

Theorem 1 (UC-Security)**.** *Let $H : \{0,1\}^* \to \{0,1\}^\lambda$ be a hash function modelled as a random oracle, and let $(\mathcal{P}, \mathcal{V})$ a zero-knowledge proof system, then Fulgor UC-realizes the ideal functionality $\mathcal{F}$ defined in Figure 5 in the $(\mathcal{F}_{\mathsf{anon}}, \mathcal{F}_{\mathsf{B}})$-hybrid model.*

4.5 System Discussion

Compatibility with Bitcoin. We note that all of the non-standard cryptographic operations (such as NIZK proofs) happen off-chain, while the only algorithm required to be executed in the verification of the blockchain is the hash function H, which can be instantiated with SHA-256. Therefore our Multi-Hop HTLC scheme and Fulgor as a whole is fully compatible with the current Bitcoin script. Moreover, as mentioned in Section 2.1, the addition of SegWit or similar solution for the malleability issue in Bitcoin fully enables payment channels in the Bitcoin system [70].

Generality. Fulgor is general to PCNs (and not only tied to Bitcoin). Fulgor requires that: (i) openChannel allows to embed custom data (e.g., fee); (ii) conditional updates of the balance in the payment channel. As arbitrary data can be included in cryptocurrency transactions [5] and most PCNs support, among others, the HTLC contract, Fulgor can be used in many other PCNs such as Raiden, a PCN for Ethereum [13].

Support for Bidirectional Channels. Fulgor can be easily extended to support bidirectional payment channels and only two minor changes are required. First, the payment information must include the direction requested at each payment channel. Second, the capacity of a channel $c_{\langle u_L,u_R\rangle}$ is a tuple of values (L, R, T) where L denotes the current balance for u_L, R is the current balance of u_R and T is the total capacity of the channel. A payment from left to right for value v is possible if $L \geq v$ and $R + v \leq T$. In such case, the tuple is updated to $(L - v, R + v, T)$. A payment from right to left is handled correspondingly.

5 Non-blocking Payments in PCNs

In this section, we discuss how to handle concurrent payments in a non-blocking manner. In other words, how to guarantee that at least one payment out of a set of concurrent payments terminates.

In the following, we start with an impossibility result that dictates the design of Rayo, our protocol for non-blocking payments. Then, we describe the modifications required in the ideal world functionality and Fulgor to achieve them. Finally, we discuss the implications of these modifications in terms of privacy properties.

5.1 Concurrency vs Privacy

We show that achieving non-blocking progress requires a global state associated to each of the payments. Specifically, we show that we cannot provide *disjoint-access parallelism* and non-blocking progress for PCNs. Formally, a PCN implementation is *disjoint-access parallel* if for any two payments channels e_i, e_j, *channel-state* $(\mathrm{e}_i) \cap$ *channel-state* $(\mathrm{e}_j) = \emptyset$.

Lemma 2. *There does not exist any strictly serializable disjoint-access parallel implementation for the payment channels problem that provides non-blocking progress.*

We defer to Appendix D for a proof sketch. Having established this inherent cost to concurrency and privacy, we model global state by a Txid field attached to each of the payments. We remark that this Txid, however, allows an adversary to reduce the set of possible senders and receivers for the payment, therefore inevitably reducing the privacy guarantees, as we discuss in Section 5.2.

5.2 Ideal World Functionality

Here, we show how to modify the ideal functionality $\mathcal{F}$, as described in Section 3.2, to account for the changes to achieve non-blocking progress in any PCN. First, a single identifier Txid (as opposed to independently sampled h_i) is used for all the payment channels in the path $(c_{\langle u_0,u_1\rangle}, \ldots, c_{\langle u_n,u_{n+1}\rangle})$. Second, $\mathcal{F}$ no longer aborts a payment simply when no capacity is left in a payment channel. Instead, $\mathcal{F}$ queues the payment if its Txid is higher than the current in-flight payment, or aborts it the Txid is lower. We detail the modified ideal functionality in Appendix C.

Discussion. Here, we discuss how the modified ideal world definition captures the security and privacy notions of interest as described in Section 3.1. In particular, it is easy to see that the notions of *balance security* and *value privacy* are enforced along the same lines. However, the leakage of the same payment identifier among all intermediate users in the payment path, reduces the possible set of sender and receivers to the actual sender and receiver

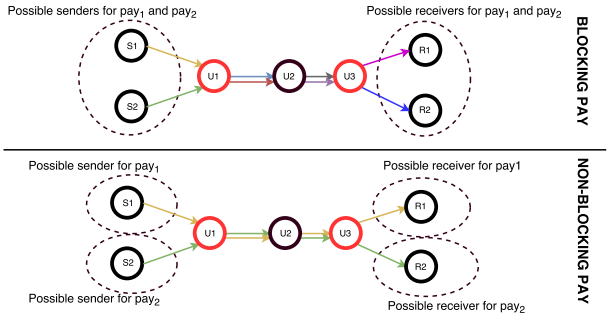

Figure 9: Illustrative example of tradeoff between concurrency and privacy. Each node represents a user: black nodes are honest and red are byzantine. In both cases, we assume two concurrent payments: S_1 pays R_1 and S_2 pays R_2 through the path U_1, U_2, U_3. The color of the arrow denotes the payment identifier. Dashed ellipses denote the anonymity set for each case.

for such payment, thereby breaking relationship anonymity. Therefore, there is an inherent tradeoff between how to handle concurrent payments (blocking or non-blocking) and the anonymity guarantees.

An illustrative example of this tradeoff is shown in Figure 9. It shows how two simultaneous payments $\mathsf{pay}_1((c_{\langle S_1,U_1\rangle}, c_{\langle U_1,U_2\rangle}, c_{\langle U_2,U_3\rangle}, c_{\langle U_3,R_1\rangle}), v)$ and $\mathsf{pay}_2((c_{\langle S_2,U_1\rangle}, c_{\langle U_1,U_2\rangle}, c_{\langle U_2,,U_3\rangle}, c_{\langle U_3,R_2\rangle}), v)$ are handled depending on whether concurrent payments are handled in a blocking or non-blocking fashion. We assume that both payments can successfully finish in the current PCN and that both payments transfer the same payment amount v, as otherwise relationship anonymity is trivially broken.

For the case of blocking payments, each intermediate user u_j observes an independently chosen identifier Txid_{ij} for each payment pay_i. Therefore, the attacker is not able to correlate the pair $(\mathsf{Txid}_{11}, \mathsf{Txid}_{21})$ (i.e., view of U_1) with the pair $(\mathsf{Txid}_{13}, \mathsf{Txid}_{23})$ (i.e., view of U_3). It follows that for a pay operation issued by any node, say S_1, the set of possible receivers that the adversary observes is $\{R_1, R_2\}$.

However, when the concurrent payments are handled in a non-blocking manner, the adversary observes for pay_1 that $\mathsf{Txid}_{11} = \mathsf{Txid}_{13}$. Therefore, the adversary can trivially derive that the only possible receiver for a pay initiated by S_1 is R_1.

5.3 Rayo: Our Construction

Building Blocks. We require the same building blocks as described in Section 4.1 and Section 4.2. The only difference is that the channel's state between two users is now defined as *channel-state* := $(cur_{(u_i,u_j)}[\,], Q_{(u_i,u_j)}[\,], \mathsf{cap}_{(u_i,u_j)})$, where *cur* denotes an array of payments currently using (part of) the capacity available at the payment channel; Q denotes the array of payments waiting for enough capacity at the payment channel, and cap denotes the current capacity value of the payment channel.

Operations. The openChannel and closeChannel operations remain as described in Section 4.3. However, the pay operation has to be augmented to ensure non-blocking payments. We have described the additional actions in light blue pseudocode in Figures 7 and 8.

In the following, we informally describe these additional actions required for the pay operation. In a nutshell, when a payment reaches an intermediate user in the path, several events can be triggered. The simplest case is when the corresponding payment channel is not saturated yet (i.e., enough capacity is left for the payment to succeed). The user accepts the payment and simply stores its information in *cur* as an in-flight payment.

The somewhat more interesting case occurs when the payment channel is saturated. This means that (possibly several) payments have been already gone through the payment channel. In this case, the simplest solution is to abort the new payment, but this leads to deadlock situations. Instead, we ensure that deadlocks do not occur by leveraging the total order of payment identifiers: If the new payment identifier (Txid) is higher than any of the payment identifiers currently active in the payment channel (i.e., included in *cur* [ ]), the payment identified by Txid is stored in Q. In this manner, if any of the currently active payments are aborted, a queued payment (Txid^*) can be recovered from Q and reissued towards the receiver. On the other hand, if Txid is lower than every identifier for currently active payments, the payment identified by Txid is directly aborted as it would not get to complete in the presence of a concurrent payment with higher identifier in the PCN.

5.4 Analysis and System Discussion

Security and Privacy Analysis. In the following, we state the security and privacy results for Rayo when handling payments in a non-blocking manner. We prove our results in the $(\mathcal{F}_{\mathsf{anon}}, \mathcal{F}_{\mathsf{B}})$-hybrid model. In other words, Theorem 2 holds for any UC-secure realization of $\mathcal{F}_{\mathsf{anon}}$ and $\mathcal{F}_{\mathsf{B}}$ (analysis in Appendix A).

Theorem 2 (UC-Security)**.** *Let $H : \{0,1\}^* \to \{0,1\}^\lambda$ be a hash function modelled as a random oracle, and let $(\mathcal{P}, \mathcal{V})$ a zero-knowledge proof system, then Rayo UC-realizes the ideal functionality $\mathcal{F}$ described in Figure 10 in the $(\mathcal{F}_{\mathsf{anon}}, \mathcal{F}_{\mathsf{B}})$-hybrid model.*

System Discussion. Rayo is compatible with Bitcoin, can be generally applicable to PCN and supports bidirectional payment channels similar to Fulgor. Moreover, the Rayo protocol provides non-blocking progress. Specifically, Rayo ensures that some payment successfully terminates in every execution. Intuitively, this is because any two conflicting payments can necessarily be ordered by their respective unique identifier: the highest payment identifier is deterministically identified and terminates successfully while the lower priority payment *aborts*.

5.5 Fulgor vs Rayo

In this work, we characterize the tradeoff between the two protocols presented in this work. As shown in Table 1, both protocols guarantee crucial security and correctness properties such as balance security and serializability. By design, Rayo is the only protocol that ensures non-blocking progress. Finally, regarding privacy, we aimed at achieving the strongest privacy possible. However, although both protocols guarantee value privacy, we have shown that it is impossible to simultaneously achieve non-blocking progress and strong anonymity. Therefore, Fulgor achieves strong anonymity while Rayo achieves non-blocking progress at the cost of weakening the anonymity guarantees. We note nevertheless that Rayo provides relationship anonymity only if none of the intermediate nodes is compromised. Intuitively, Rayo provides this (weaker) privacy guarantee because it still uses Multi-Hop HTLC as Fulgor.

Table 1: Comparison between Fulgor and Rayo.

	Fulgor	**Rayo**
Balance security	●	●
Serializability	●	●
Non-blocking progress	○	●
Value Privacy	●	●
Anonymity	●	◐

6 Performance Analysis

In this section, we first evaluate the performance of Fulgor. Finally, we describe the overhead required for Rayo.

We have developed a proof-of-concept implementation in Python to evaluate the performance of Fulgor. We interact with the API of *lnd* [1], the recently released Lightning Network implementation, We use *listchannels* to extract the current capacity of an open payment channel, *listpeers* to extract the list of public keys from other users in the network, and *getinfo* to extract the user's own public key. We have instantiated the hash function with SHA-256. We have implemented the Multi-Hop HTLC using a python-based implementation of ZK-Boo [63] to create the zero-knowledge proofs. We set ZK-Boo to use SHA-256, 136 rounds to achieve a soundness error of the proofs of 2^{-80}, and a witness of 32 bytes as in [69].

Implementation-level Optimizations. During the protocol description, we have assumed that the sender creates a different anonymous communication channel with each intermediate user. In our implementation, however, we use Sphinx [31] to create a single anonymous communication channel between sender and receiver, where intermediate nodes are the intermediate users in the path. Sphinx allows to send the required payment information to each intermediate user while obfuscating the information intended for other users in the path and the actual length of the path by padding the forwarded data. This optimization has been discussed in the bitcoin community and implemented in the current release of *lnd* [9].

Testbed. We have simulated five users and created a linear structure of payment channels: user i has payment channels open only with user $i-1$ and user

$i+1$, user 0 is the sender, and user 4 is the receiver of the pay operation. We run each of the users in a separated virtual machine with an Intel Core i7 3.1 GHz processor and 2 GB RAM. The machines are connected in a local network with a mean latency of 111.5 milliseconds. For our experiments, we assume that each user has already opened the corresponding payment channels and got the public verification key of each other user in the PCN. As this is a one time setup operation, we do not account for it in our experiments.

Performance. We have first executed the payment operation available in the *lnd* software, which uses the HTLC-based payment as the contract for conditional updates in a payment channel. We observe that a (non-private) pay operation over a path with 5 users takes 609 ms and so needs Fulgor. Additionally, the Sdr must run the $\mathsf{Setup}_{\mathrm{HTLC}}(n+1)$ protocol, increasing thereby her computation time. Moreover, the Sdr must send the additional information corresponding to the Multi-Hop HTLC contract (i.e., $(x_i, y_i, y_{i+1}, \pi_i)$) to each intermediate user, which adds communication complexity.

The sender requires 309 ms to compute the proof π_i for each of the intermediate users. Each proof is of size 1.65 MB. Finally, each intermediate user requires 130 ms to verify π_i. We focus on the zero-knowledge proofs as they are the most expensive operation.

Therefore, the total computation overhead is 1.32 seconds (*lnd* pay and Multi-Hop HTLC) and the total communication overhead is less than 5 MB (3 zero-knowledge proofs plus the tuple of small-size values (x_i, y_i, y_{i+1}) per intermediate user). We observe that previous proposal [69] required around 10 seconds to compute only a single zero-knowledge proof. In contrast, the pay operation in Fulgor requires less than 2 seconds of computation and to communicate less than 5 MB among the users in the path for the complete payment operation, which demonstrates the practicality of Fulgor.

Scalability. In order to test the scalability of the pay operation in Fulgor, we have studied the running time and communication overhead required by each of the roles in a payment (i.e., sender, receiver, and intermediate user). Here, we take into account that Sphinx requires to pad the forwarded messages to the maximum path length. In the absence of widespread PCN in practice, we set the maximum path length to 10 in our test, as suggested for similar payment networks such as the Ripple credit network [49].

Regarding the computation time, the sender requires 3.09 seconds to create π_i for each intermediate user. However, this computation time can be improved if different π_i are calculated in parallel taking advantage of current multi-core systems. Each intermediate user requires 130 ms as only has to check the contract for payment channels with successor and predecessor user in the path. Finally, the receiver incurs in few ms as she only has to check whether a given value is the correct pre-image of a given hash value.

Regarding communication overhead, the sender must create a message with 10 proofs of knowledge and other few bytes associated to the contract for each intermediate payment channel. So in total, the sender must forward 17MB approximately. As Sphinx requires padded messages at each node to ensure anonymity, every intermediate user must forward a message of the same size.

In summary, these results show that even with an unoptimized implementation, a payment with 10 intermediate users takes less than 5 seconds and require a communication overhead of approximately 17MB at each intermediate user. Therefore, Fulgor induces a relatively small overhead while enabling payments between any two users in the PCN and has the potential to be deployed as a PCN with a growing base of users performing payments with even 10 intermediate users in a matter of few seconds, a result in line with other privacy preserving payment systems [54, 49].

Non-blocking payments (Rayo). Given their similar definitions, the performance evaluation for Fulgor carries over to Rayo. Additionally, the management of non-blocking payments requires that intermediate users maintain a list (cur) of current in-flight payments and a queue (Q) of payments waiting to be forwarded when capacity is available. The management of these data structures requires a fairly small computation overhead. Moreover, the number of messages to be stored in these data structures according to the specification of Rayo is clearly linear in the length of the path. Specifically, a payment involving a path of length $k \in \mathbb{N}$ incurs $\mathrm{O(c{\cdot}k)}$ message complexity, where c is bounded by the total of concurrent conflicting payments.

7 Related Work

Payment channels were first introduced by the Bitcoin community [2] and since then, several extensions have been proposed. Decker and Wattenhofer [32] describe bidirectional payment channels [32]. Lind et al. [47] leverage trusted platform modules to use a payment channel without hindering compatibility with Bitcoin. However, these works focus on a single payment channel and their extension to support PCNs remain an open challenge.

TumbleBit [41] and Bolt [40] propose off-chain path-based payments while achieving sender/receiver anonymity in Tumblebit and payment anonymity in Bolt. However, these approaches are restricted to single hop payments, and it is not clear how to extend them to account for generic multi-hop PCNs and provide the privacy notions of interest, as achieved by Fulgor and Rayo.

The Lightning Network [60] has emerged as the most prominent proposal for a PCN in Bitcoin. Other PCNs such as Thunder [19] and Eclair [10] for Bitcoin and Raiden [13] for Ethereum are being proposed as slight modifications of the Lightning Network. Nevertheless, their use of HTLC leaks a common identifier per payment, thereby reducing the anonymity guarantees as we described in this work. Moreover, current proposals lack a non-blocking solution for concurrent payments. Fulgor and Rayo, instead, rely on Multi-Hop HTLC to overcome the linkability issue with HTLC. They provide a tradeoff between non-blocking progress and anonymity.

Recent works [54, 49] propose privacy definitions for credit networks, a payment system that supports multi-hop payments similar to PCNs. Moreover, privacy preserving protocols are described for both centralized [54] and decentralized credit networks [49]. However, credit networks differ from PCNs in that they do not require to ensure accountability against an underlying blockchain. This requirement reduces the set of cryptographic operations available to design a PCN. Nevertheless, Fulgor and Rayo provide similar privacy guarantees as credit networks even under those restrictions.

Miller et al [53] propose a construction for payment channels to reduce the time that funds are locked at intermediate payment channels (i.e., collateral cost), an interesting problem but orthogonal to our work. Moreover, they formalize their construction for multi-hop payments as an ideal functionality. However, they focus on collateral cost and do not discuss privacy guarantees, concurrent payments are handled in a blocking manner only, and their construction relies on smart contracts available on Ethereum that are incompatible with the current Bitcoin scripting system.

Towns proposed [69] a variation of the HTLC contract, based on zk-SNARKs, to avoid its linkability problem among payment channels in a path. However, the Bitcoin community has not adopted this approach due to its inefficiency. In this work, we revisit this solution with a formal protocol with provable security and give an efficient instantiation based on ZK-Boo [38].

8 Conclusion

Permisionless blockchains governed on global consensus protocols face, among others, scalability issues in catering a growing base of users and payments. A burgeoning approach to overcome this challenge consists of PCNs and recent efforts have derived in the first yet alpha implementations such as the Lightning Network [60] in Bitcoin or Raiden [13] in Ethereum. We are, however, only scratching the surface as many challenges such as liquidity, network formation, routing scalability, concurrency or privacy are yet to be thoroughly studied.

In this work, we lay the foundations for privacy and concurrency in PCNs. In particular, we formally define in the Universal Composability framework two modes of operation for PCNs attending to how concurrent payments are handled (blocking versus non-blocking). We provide formally proven instantiations (Fulgor and Rayo) for each, offering a tradeoff between non-blocking progress and anonymity. Our evaluation results demonstrate that is feasible to deploy Fulgor and Rayo in practice and can scale to cater a growing number of users.

Acknowledgments. We thank the anonymous reviewers for their helpful reviews, and Ivan Pryvalov for providing his python-based implementation of ZK-Boo.

This work is partially supported by a Intel/CERIAS research assistantship, and by the National Science Foundation under grant CNS-1719196.

This research is based upon work supported by the German research foundation (DFG) through the collaborative research center 1223 and by the state of Bavaria at the Nuremberg Campus of Technology (NCT). NCT is a research cooperation between the Friedrich-Alexander-University Erlangen-Nürnberg (FAU) and the Technische Hochschule Nürnberg Georg Simon Ohm (THN).

References

[1] Alpha release of the lightning network daemon. Blog entry. http://lightning.community/release/software/lnd/lightning/2017/01/10/lightning-network-daemon-alpha-release/.

[2] Bitcoin wiki: Bitcoin contract. https://en.bitcoin.it/wiki/Contract.

[3] Bitcoin wiki: Bitcoin scalability faq. https://en.bitcoin.it/wiki/Scalability_FAQ.

[4] Bitcoin wiki: Block size limit controversy. https://en.bitcoin.it/wiki/Block_size_limit_controversy.

[5] Bitcoin wiki: Op_return. https://en.bitcoin.it/wiki/OP_RETURN.

[6] Bitcoin wiki: Payment channels. https://en.bitcoin.it/wiki/Payment_channels.

[7] Bitcoin wiki: Timelock. https://en.bitcoin.it/wiki/Timelock.

[8] Bitcoin wiki: Transaction malleability. https://en.bitcoin.it/wiki/Transaction_Malleability.

[9] Bolt #4: Onion routing protocols. https://github.com/lightningnetwork/lightning-rfc/blob/master/04-onion-routing.md.

[10] Eclair implementation of the lightning network. https://github.com/ACINQ/eclair.

[11] Lightning network daemon. Github implementation. https://github.com/LightningNetwork/lnd.

[12] Lightning protocol reference implementation. Github implementation. https://github.com/ElementsProject/lightning.

[13] Raiden network. Project's website. http://raiden.network/.

[14] Reliable transaction submission. Ripple protocol's documentation. https://ripple.com/build/reliable-transaction-submission/.

[15] Ripple protocol. Project's website. https://ripple.com/.

[16] Segregated witness adoption. Blog entry. https://bitcoincore.org/en/segwit_adoption/.

[17] Stellar protocol. Project's website. https://www.stellar.org/.

[18] Stress test prepares visanet for the most wonderful time of the year. Blog entry. http://www.visa.com/blogarchives/us/2013/10/10/stress-test-prepares-visanet-for-the-most-wonderful-time-of-the-year/index.html.

[19] Thunder network. Project's website. https://github.com/blockchain/thunder.

[20] Alpern, B., and Schneider, F. B. Defining liveness. *Inf. Process. Lett. 21*, 4 (Oct. 1985), 181–185.

[21] Androulaki, E., Karame, G. O., Roeschlin, M., Scherer, T., and Capkun, S. Evaluating user privacy in bitcoin. Financial Cryptography and Data Security 2013.

[22] Atlas, K. The inevitability of privacy in lightning networks. Blog entry. https://www.kristovatlas.com/the-inevitability-of-privacy-in-lightning-networks/.

[23] Attiya, H., and Welch, J. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics.* John Wiley & Sons, 2004.

[24] Backes, M., Kate, A., Manoharan, P., Meiser, S., and Mohammadi, E. Anoa: A framework for analyzing anonymous communication protocols. In *IEEE 26th Computer Security Foundations Symposium* (2013).

[25] Barber, S., Boyen, X., Shi, E., and Uzun, E. Bitter to better. how to make Bitcoin a better currency. Financial Cryptography and Data Security 2012.

[26] Camenisch, J., and Lysyanskaya, A. A formal treatment of onion routing. In *Advances in Cryptology—CRYPTO* (2005).

[27] Canetti, R. Universally composable security: A new paradigm for cryptographic protocols. In *"FOCS'01"*.

[28] Cristian, F., Aghili, H., and Strong, H. R. Approximate clock synchronization despite omission and performance faults and processor joins. In *Proceedings of the 16th International Symposium on Fault-Tolerant Computing* (July 1986).

[29] Dandekar, P., Goel, A., Govindan, R., and Post, I. Liquidity in credit networks: a little trust goes a long way. In *ACM Conference on Electronic Commerce* (2011).

[30] Dandekar, P., Goel, A., Wellman, M. P., and Wiedenbeck, B. Strategic formation of credit networks. In *WWW* (2012).

[31] Danezis, G., and Goldberg, I. Sphinx: A compact and provably secure mix format. In *30th IEEE Symposium on Security and Privacy (S&P 2009)*.

[32] Decker, C., and Wattenhofer, R. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Stabilization, Safety, and Security of Distributed Systems* (2015).

[33] Dryja, T. Unlinkable outsourced channel monitoring. (Talk transcript) `https://diyhpl.us/wiki/transcripts/scalingbitcoin/milan/unlinkable-outsourced-channel-monitoring/`.

[34] Fischer, M. J., Lynch, N. A., and Paterson, M. S. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (Apr. 1985), 374–382.

[35] Friedenbach, M., BtcDrak, Dorier, N., and kinoshitajona. Bip 68: Relative lock-time using consensus-enforced sequence numbers. `https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki`.

[36] Fugger, R. Money as ious in social trust networks & a proposal for a decentralized currency network protocol. Technical Report, 2004. `http://archive.ripple-project.org/decentralizedcurrency.pdf`.

[37] Ghosh, A., Mahdian, M., Reeves, D. M., Pennock, D. M., and Fugger, R. Mechanism design on trust networks. In *WINE'07*.

[38] Giacomelli, I., Madsen, J., and Orlandi, C. Zkboo: Faster zero-knowledge for boolean circuits. In *USENIX Security* (2016).

[39] go1111111 (pseudonym). Idea to improve lightning network. Forum post. `https://bitcointalk.org/index.php?topic=1134319.0`.

[40] Green, M., and Miers, I. Bolt: Anonymous payment channels for decentralized currencies. In *CCS* (2017).

[41] Heilman, E., Alshenibr, L., Baldimtsi, F., Scafuro, A., and Goldberg, S. TumbleBit: An untrusted bitcoin-compatible anonymous payment hub. In *NDSS* (2017).

[42] Herlihy, M., and Shavit, N. On the nature of progress. In *OPODIS* (2011), pp. 313–328.

[43] Herrera-Joancomartí, J., and Pérez-Solà, C. Privacy in bitcoin transactions: New challenges from blockchain scalability solutions. In *Modeling Decisions for Artificial Intelligence* (2016).

[44] Kosba, A., Miller, A., Shi, E., Wen, Z., and Papamanthou, C. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE S&P* (2016).

[45] Koshy, P., Koshy, D., and McDaniel, P. An analysis of anonymity in bitcoin using p2p network traffic. In *Financial Cryptography and Data Security* (2014).

[46] Lamport, L., and Fischer, M. Byzantine generals and transaction commit protocols. Tech. Rep. 62, SRI International, Apr. 1982.

[47] Lind, J., Eyal, I., Pietzuch, P. R., and Sirer, E. G. Teechan: Payment channels using trusted execution environments. http://arxiv.org/abs/1612.07766.

[48] Lopp, J. Lightning's balancing act: Challenges face bitcoin's scalability savior. Blog entry. http://www.coindesk.com/lightning-technical-challenges-bitcoin-scalability/.

[49] Malavolta, G., Moreno-Sanchez, P., Kate, A., and Maffei, M. SilentWhispers: Enforcing security and privacy in credit networks. In *NDSS* (2017).

[50] McCorry, P., Möser, M., Shahandashti, S. F., and Hao, F. Towards bitcoin payment networks. In *Australasian Conference Information Security and Privacy* (2016).

[51] Meiklejohn, S., and Orlandi, C. Privacy-enhancing overlays in bitcoin. In *BITCOIN* (2015).

[52] Meiklejohn, S., Pomarole, M., Jordan, G., Levchenko, K., McCoy, D., Voelker, G. M., and Savage, S. A fistful of bitcoins: Characterizing payments among men with no names. In *IMC* (2013).

[53] Miller, A., Bentov, I., Kumaresan, R., and McCorry, P. Sprites: Payment channels that go faster than lightning. *CoRR abs/1702.05812* (2017).

[54] Moreno-Sanchez, P., Kate, A., Maffei, M., and Pecina, K. Privacy preserving payments in credit networks. In *NDSS* (2015).

[55] Moreno-Sanchez, P., Modi, N., Songhela, R., Kate, A., and Fahmy, S. Mind your credit: Assessing the health of the ripple credit network. *CoRR abs/1706.02358* (2017).

[56] Moreno-Sanchez, P., Zafar, M. B., and Kate, A. Listening to whispers of ripple: Linking wallets and deanonymizing transactions in the ripple network. In *PETS* (2016).

[57] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[58] Papadimitriou, C. H. The serializability of concurrent database updates. *J. ACM 26* (1979), 631–653.

[59] Pfitzmann, A., and Hansen, M. Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management – a consolidated proposal for terminology, 2008.

[60] Poon, J., and Dryja, T. The bitcoin lightning network: Scalable off-chain instant payments. Technical Report. https://lightning.network/lightning-network-paper.pdf.

[61] Post, A., Shah, V., and Mislove, A. Bazaar: Strengthening user reputations in online marketplaces. In *NSDI* (2011).

[62] Prihodko, P., Zhigulin, S., Sahno, M., and Ostrovskiy, A. Flare: An approach to routing in lightning network. http://bitfury.com/content/5-white-papers-research/whitepaper_flare_an_approach_to_routing_in_lightning_network_7_7_2016.pdf.

[63] Pryvalov, I. pyZKBoo++ implementation. Project's website. https://sites.google.com/view/pyzkboopp/home.

[64] Reid, F., and Harrigan, M. An analysis of anonymity in the bitcoin system. In *Security and Privacy in Social Networks* (2013).

[65] Russell, R. Reaching the ground with lightning. Technical Report. http://ozlabs.org/~rusty/ln-deploy-draft-01.pdf.

[66] Spagnuolo, M., Maggi, F., and Zanero, S. BitIodine: Extracting intelligence from the bitcoin network. In *Financial Cryptography and Data Security* (2014).

[67] Torpey, K. Brock pierce: Bitcoin's scalability issues are a sign of its success. Blog entry.

https://bitcoinmagazine.com/articles/brock-pierce-bitcoin-s-scalability-issues-are-a-sign-of-its-success-1459867433/.

[68] Torpey, K. Does the lightning network threaten bitcoin?s censorship resistance? Blog entry. https://bitcoinmagazine.com/articles/does-the-lightning-network-threaten-bitcoin-s-censorship-resistance-1461953131/.

[69] Towns, A. Better privacy with SNARKs. Mailing List. https://lists.linuxfoundation.org/pipermail/lightning-dev/2015-November/000309.html.

[70] van Wirdum, A. Segwit or not, bitfury is getting ready for lightning with successful bitcoin main net test. Blog entry. https://bitcoinmagazine.com/articles/segwit-or-not-bitfury-ready-lightning-successful-bitcoin-main-net-test/.

[71] Viswanath, B., Mondal, M., Gummadi, K. P., Mislove, A., and Post, A. Canal: Scaling social network-based sybil tolerance schemes. In *EuroSys* (2012).

[72] Wikström, D. A universally composable mix-net. In *Theory of Cryptography Conference* (2004), M. Naor, Ed.

A Security Analysis

Our proof strategy consists of the description of a simulator $\mathcal{S}$ that handles users corrupted by the attacker and simulates the real world execution protocol while interacting with the ideal functionality $\mathcal{F}$. The simulator $\mathcal{S}$ spawns honest users at adversarial will and impersonates them until the environment $\mathcal{E}$ makes a corruption query on one of the users: At this point $\mathcal{S}$ hands over to $\mathcal{A}$ the internal state of the target user and routes all of the subsequent communications to $\mathcal{A}$, who can reply arbitrarily. For operations exclusively among corrupted users, the environment does not expect any interaction with the simulator. Similarly, communications exclusively among honest nodes happen through secure channels and therefore the attacker does not gather any additional information other than the fact that the communication took place. For simplicity, we omit these operations in the description of our simulator. The random oracle H is simulated by $\mathcal{S}$ via lazy-sampling. The operations to be simulated for a PCN are described in the following.

$\mathsf{openChannel}(c_{\langle u_1,u_2\rangle}, \beta, t, f)$: Let u_1 be the user that initiates the request. We analyze two possible cases:

1. *Corrupted u_1:* $\mathcal{S}$ receives a $(c_{\langle u_1,u_2\rangle}, \beta, t, f)$ request from the adversary on behalf of u_1 and initiates a two-user agreement protocol with $\mathcal{A}$ to convey upon a local fresh channel identifier $c_{\langle u_1,u_2\rangle}$. If the protocol successfully terminates, $\mathcal{S}$ sends $(\mathsf{open}, c_{\langle u_1,u_2\rangle}, \beta, t, f)$ to $\mathcal{F}$, which eventually returns $(c_{\langle u_1,u_2\rangle}, h)$.

2. *Corrupted u_2:* $\mathcal{S}$ receives a message $(c_{\langle u_1,u_2\rangle}, v, t, f)$ from $\mathcal{F}$ engages $\mathcal{A}$ in a two-user agreement protocol on behalf of u_1 for the opening of the channel. If the execution is successful, $\mathcal{S}$ sends an accepting message to $\mathcal{F}$ which returns $(c_{\langle u_1,u_2\rangle}, h)$, otherwise it outputs $\perp$.

If the opening was successful the simulator initializes an empty list $\mathcal{L}_{c_{\langle u_1,u_2\rangle}}$ and appends the value $(h, v, \perp, \perp)$.

$\mathsf{closeChannel}(c_{\langle u_1,u_2\rangle}, v)$: Let u_1 be the user that initiates the request. We distinguish two possible scenarios:

1. *Corrupted u_1:* $\mathcal{S}$ receives a closing request from the adversary on behalf of u_1, then it fetches $\mathcal{L}_{c_{\langle u_1,u_2\rangle}}$ for some value (h, v, x, y). If such a value does not exist then it aborts. Otherwise it sends $(\mathsf{close}, c_{\langle u_1,u_2\rangle}, h)$ to $\mathcal{F}$.

2. *Corrupted u_2:* $\mathcal{S}$ receives $(c_{\langle u_1,u_2\rangle}, h, \perp)$ from $\mathcal{F}$ and simply notifies $\mathcal{A}$ of the closing of the channel $c_{\langle u_1,u_2\rangle}$.

$\mathsf{pay}((c_{\langle u_0,u_1\rangle}, \ldots, c_{\langle u_n,u_{n+1}\rangle}), v)$: Since the specifications of the protocol differ depending on whether a user is a sender, a receiver or an intermediate node of a payment, we consider the cases separately.

1. *Sender:* In order to initiate a payment, the adversary must provide each honest user u_i involved with a message m_i that the simulator parses as $(c_{\langle u_{i-1},u_i\rangle}, c_{\langle u_i,u_{i-1}\rangle}, x_i, y_i, y_{i+1},$

π_i, v_i, t_i, t_{i+1}), also the receiver of the payment u_{n+1} (in case it is not corrupted) is notified with some message ($c_{\langle u_n, u_{n+1}\rangle}, x_n$, y_n, v, t_n). For each intermediate honest user u_i, the simulator checks whether $t_i \geq t_{i+1}$ and $\mathcal{V}((H, y_i, y_{i+1}, x_i), \pi_i) = 1$. If the conditions hold, $\mathcal{S}$ sends to $\mathcal{F}$ the tuple ($\mathsf{pay}, v_i, (c_{\langle u_{i-1}, u_i\rangle}$, $c_{\langle u_i, u_{i+1}\rangle}), t_{i-1}, t_i$), whereas for the receiver (in case it is honest) sends ($\mathsf{pay}, v, c_{\langle u_n, u_{n+1}\rangle}, t_n$) if $y_n = H(x_n)$, otherwise it aborts. For each intermediate user u_i the simulator confirms the payment only when receives from the user u_{i+1} an x such that $H(x_i \oplus x) = y_i$. If $\mathcal{A}$ outputs a value x^* such that $H(x^*) = y_{i+1}$ but $H(x_i \oplus x^*) \neq y_i$ then $\mathcal{S}$ aborts the simulation. If the receiver is honest then the simulator confirms the payment if the amount v corresponds to what agreed with the sender and if $H(x_n) = y_n$. If the payment is confirmed the entry $(h_i, v^* - v_i, x_i \oplus x, y_i)$ is added to $\mathcal{L}_{c_{\langle u_{i-1}, u_i\rangle}}$, where $(h_i^*, v^*, \cdot, \cdot)$ is the entry of $\mathcal{L}_{c_{\langle u_{i-1}, u_i\rangle}}$ with the lowest v^*, and the same happens for the receiver.

2. *Receiver:* $\mathcal{S}$ receives some $(h, c_{\langle u_n, u_{n+1}\rangle}, v, t_n)$ from $\mathcal{F}$, then it samples a random $x \in \{0,1\}^\lambda$ and returns to $\mathcal{A}$ the tuple $(x, H(x), v)$. If $\mathcal{A}$ returns a string $x' = x$, then $\mathcal{S}$ returns $\top$ to $\mathcal{F}$, otherwise it sends $\bot$.

3. *Intermediate user:* $\mathcal{S}$ is notified that a corrupted user is involved in a payment with a message of the form $(h_i, h_{i+1}$, $c_{\langle u_{i-1}, u_i\rangle}, c_{\langle u_i, u_i+1\rangle}, v, t_{i-1}, t_i)$ by $\mathcal{F}$. $\mathcal{S}$ samples an $x \in \{0,1\}^\lambda$ and an $x' \in \{0,1\}^\lambda$ and runs the simulator of the zero-knowledge scheme to obtain the proof π over the statement $(H, H(x \oplus x'), H(x'), x)$. The adversary is provided with the tuple $(c_{\langle u_{i-1}, u_i\rangle}, c_{\langle u_i, u_{i-1}\rangle}, x, H(x \oplus x'), H(x'), \pi, v, t_{i-1}, t_i)$ via an anonymous channel. If $\mathcal{A}$ outputs a string $x'' = x \oplus x'$, then $\mathcal{S}$ aborts the simulation. At some point of the execution the simulator is queried again on (h_i, h_{i+1}), then it sends x' to $\mathcal{A}$ on behalf of u_{i+1}. If $\mathcal{A}$ outputs a string $z = x \oplus x'$ the simulator sends $\top$ to $\mathcal{F}$ and appends $(h_i, v^* - v, z, H(z))$ to $\mathcal{L}_{c_{\langle u_{i-1}, u_i\rangle}}$, where $(h_i^*, v^*, \cdot, \cdot)$ is the entry of $\mathcal{L}_{c_{\langle u_{i-1}, u_i\rangle}}$ with the lowest v^*. The simulator sends $\bot$ otherwise. Note that we consider the simpler case where a single node in the payment is corrupted. However this can be easily extended to the more generic case by book-keeping the values of h_i and choosing the corresponding the pre-images x and x' consistently. The rest of the simulation is unchanged.

Analysis. Since the simulation runs only polynomially-bounded algorithms it is easy to see that the simulation is efficient. We now argue that the view of the environment in the simulation is indistinguishable from the execution of the real-world protocol. For the moment, we assume that the simulation never aborts, then we separately argue that the probability of the simulator to abort is negligible. For the $\mathsf{openChannel}$ and $\mathsf{closeChannel}$ algorithms the indistinguishability argument is trivial. On the other hand for the payment we need a more sophisticated reasoning. Consider first the scenario where the sender is corrupted: In this case the simulation diverges form the the original protocol since each multi-hop payment is broke down into separate single-hop payments. Note that the off-chain communication mimics exactly the real-world protocol (as long as $\mathcal{S}$ does not abort): Each node u_i that is not the receiver confirms the transaction to $\mathcal{F}$ only if it learns a valid pre-image of its y_i. Since we assume that the simulation does not abort, it follows that the simulation is consistent with the fact that each honest node always returns $\top$ at this stage of the execution (i.e., the payment chain does not stop at a honest node, other than the sender). However, the values published in the blockchain could in principle diverge from what the adversary is expecting in the real execution. In fact, an entry of the real blockchain contains the values of $(x, H(x))$ corresponding to a particular payment, in addition to the information that is leaked by the ideal functionality. Therefore we have to show that the values of (x, y) that the simulator appends to $\mathcal{A}$'s view of B (in the $\mathsf{closeChannel}$ simulation) have the same distribution as in the real world. Note that those values are either selected by the adversary if the sender is corrupted (there the argument is trivial) or chosen to be $(x, H(x))$ by the simulator, for some randomly chosen $x \in \{0,1\}^\lambda$. For the latter case it is enough to observe that the following distributions are statistically close

$$\left(\left(\bigoplus_{i=1}^{n} x_i, y_1\right), \ldots, (x_n, y_n)\right) \approx ((r_1, s_1), \ldots, (r_n, s_n)),$$

where for all $i : (x_i, r_i) \leftarrow \{0,1\}^{2\cdot\lambda}$, $y_i \leftarrow H(x_i)$, and $s_i \leftarrow H(r_i)$. Note that on the left hand side of the equation the values are distributed accordingly to the real-world protocol, while on the right hand side the distribution corresponds to the simulated values. The indistinguishability follows. For the simulation of the receiver and of the intermediate users one can use a similar argument. We only need to make sure that $\mathcal{A}$ cannot interrupt a payment chain before it reaches the receiver, which is not allowed in the ideal world. It is easy to see that in that case ($\mathcal{A}$ outputs x'' such that $H(x'') = H(x \oplus x')$ before receiving x') the simulation aborts.

What is left to be shown is that the simulation aborts with at most negligible probability. Let $\mathsf{abort}_\mathrm{s}$ the event that $\mathcal{S}$ aborts in the simulation of the sender and let $\mathsf{abort}_\mathrm{i}$ be the event that $\mathcal{S}$ aborts in the simulation of the intermediate user. By the union bound we have that $\mathsf{Pr}[\mathsf{abort}] \leq \mathsf{Pr}[\mathsf{abort}_\mathrm{s}] + \mathsf{Pr}[\mathsf{abort}_\mathrm{i}]$.

We note that in case $\mathsf{abort}_\mathrm{s}$ happens than the adversary was able to output a valid proof π_i over (H, y_i, y_{i+1}, x_i) and an x^* such that $H(x^*) = y_{i+1}$ and $H(x^* \oplus x_i) \neq y_i$. Let w be a bitstring such that $H(w) = y_{i+1}$ and $H(w \oplus x_i) = y_i$, by the soundness of the proof π_i such a string is guaranteed to exists. It follows that $H(x^* \oplus x_i) \neq H(w \oplus x_i)$ which implies that $w \neq x^*$, since H is a deterministic function. However we have that $H(x^*) = H(w)$, which implies that $w = x^*$, since $\mathcal{A}$ can query the random oracle at most polynomially-many times. This is a contradiction and therefore it must be the case that for all PPT adversaries the probability of $\mathsf{abort}_\mathrm{s}$ to happen is 0. We can now rewrite $\mathsf{Pr}[\mathsf{abort}] \leq \mathsf{Pr}[\mathsf{abort}_\mathrm{i}]$. Consider the event $\mathsf{abort}_\mathrm{i}$: In this case we have that $\mathcal{A}$, on input $(H(x \oplus x'), H(x'), x)$, is able to output some $x'' = x \oplus x'$. Note that x' is a freshly sampled value and therefore the values $H(x \oplus x')$ and $H(x')$ are uniformly distributed over the range of H. Thus the probability that $\mathcal{A}$ is able to output the pre-image of $H(x \oplus x')$ without knowing x' is bounded by a negligible function in the security parameter. It follows that $\mathsf{Pr}[\mathsf{abort}] \leq \mathsf{negl}(\lambda)$. And this concludes our proof. □

Non-Blocking Solution. The security proof for our non-blocking solution is identical to what described above, with the only exception that the ideal functionality leaks the identifier of a payment to the intermediate users. Therefore the simulator must make sure to choose the transaction identifier consistently for all of the corrupted users involved in the same payment. In addition to that, the simulator must also implement the non-blocking logic for the queueing of the payments. The rest of the argument is unchanged.

B Agreement between Two Users

In this section, we describe the protocol run by two users, u_0 and u_1, sharing a payment channel to reach *agreement* [34] on the channel's state at each point in time.

Notation and Assumptions. In this section, we follow the notation we introduced in Section 4. We assume that there is a total order between the events received by the users at a payment channel (e.g., lexicographically sorted by the hash of the corresponding payment data) and the users (e.g., lexicographically sorted by their public verification keys). Moreover, we assume that users perform the operations associated to each event as defined in our construction (see Section 4.3). Therefore, in this section we only describe the additional steps required by users to handle concurrent payments. Finally, we assume that two users sharing a payment channel, locally maintain the state of the payment channel (*channel-state*). The actual definition of *channel-state* depends on whether concurrent payments are handled in a blocking or non-blocking manner. For blocking, *channel-state* is defined as $\mathsf{cap}(c_{\langle u_0,u_1\rangle})$, where cap denotes the current capacity in the payment channel. For non-blocking, *channel-state* is defined as a tuple $\{cur[\,], Q[\,], \mathsf{cap}\}$, where *cur* denotes an array of payments currently using (part of) the capacity available at the payment channel; Q denotes the array of payments waiting for enough capacity at the payment channel.

The agreement on *channel-state* between the corresponding two users u_0 and u_1 is performed in two communication rounds. In the first round, both users exchange the set of events $\{decision_b\}$ to be applied into the *channel-state*. At the end of this first round, each user comes up with the aggregated set of events $\{decision\} := \{decision\}_0 \cup \{decision\}_1$ deterministically sorted according to the following criteria. First, the events proposed by the user

with the highest identifier are included first. Second, if several events are included in $\{decision\}_b$, they are sorted according to the following sequence: accept, abort, forward.[5] Finally, events of the same type are sorted in decreasing order by the corresponding payment identifier. These set of rules ensure that the both users can deterministically compute the same sorted version of the set $\{decision_i\}$.

Before starting the second communication round, each user applies the changes related to each event in $\{decision_i\}$ to the current *channel-state*. The mapping between each event and the corresponding actions is defined as a function $\{(decision_j, m_j)\} \leftarrow f(\{decision_i\})$. This function returns a set of tuples that indicate what events must be forwarded to which user in the payment path. Then, in the second communication round, each event $decision_j$ is sent to the corresponding user u_j (encoded in m_j). The actual implementation of the function f determines how the concurrent payments are handled. In Fulgor, we implement the function f as described in Figures 7 and 8 (black pseudocode) for blocking approach and as described in Figures 7 and 8 (light blue pseudocode) for non-blocking approach.

In the following, we denote the complete agreement protocol between two users by $\mathsf{2ProcCons}(u_0, u_1, \{decision_i\})$.

Lemma 3. $\mathsf{2ProcCons}(u_0, u_1, \{decision_i\})$ *ensures agreement on the channel-state given the set of events* $\{decision\}$.

Proof. Assume that *channel-state* is consistent between two users u_i and u_j before $\mathsf{2ProcCons}(u_0, u_1, \{decision_i\})$ is invoked. It is easy to see that both users come with the same sorted version of $\{decision_i\}$ since the sorting rules are deterministic. Moreover, for each event, the function f deterministically updates *channel-state* and returns a tuple $(m, decision)$. As the events are applied in the same order by both users, they reach agreement on the same updated *channel-state* and the same set of tuples $\{(u_k, decision_k)\}$. □

[5] Although other sequences are possible, we fix this one to ensure that the sorting is deterministic.

C Ideal World Functionality for Non-Blocking Payments

In this section, we detail the ideal world functionality for a PCN that handles concurrent payments in a non-blocking manner. We highlight in light blue the changes with respect to the ideal world functionality presented in Section 3.2 that correspond to a PCN that handles concurrent payments in a blocking manner. Moreover, we assume the same model, perform the same assumptions and use the same notation as described in Section 3.2. Additionally, we use the variable queued to track at which intermediate user the payment is queued if there is not enough capacity in her channel and the payment identifier is higher than those in-flight. Moreover, we use a list $\mathcal{W}$ to keep track of remaining hops for queued payments. Entries in $\mathcal{W}$ are of the form $((c_{\langle u_1,u_2\rangle}, \ldots, c_{\langle u_k,u_{k+1}\rangle}), v, (t_1, \ldots, t_k))$ and contain the remaining list of payment channels $(c_{\langle u_1,u_2\rangle}, \ldots, c_{\langle u_k,u_{k+1}\rangle})$, their associated timeouts $(t_1, \ldots, t_k)$ and the remaining payment value v.

For simplicity we only model unidirectional channels, although our functionality can be easily extended to support also bidirectional channels. The execution of our simulation starts with $\mathcal{F}$ querying $\mathcal{F}_{\mathsf{B}}$ to initialize it and $\mathcal{F}$ initializing itself the locally stored empty lists $\mathcal{L}, \mathcal{C}, \mathcal{W}$.

D Proof for Concurrency Lemmas

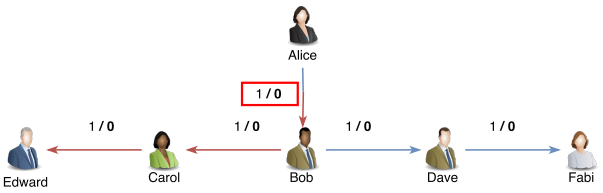

Figure 11: Execution depicting two payments:payment Txid_i from Alice to Edward and payment Txid_j from Alice to Fabi. If Alice and Bob are byzantine, they can allow both payments to be successful (while losing funds themselves).

Proof for Lemma 1. Consider an execution of two payments depicted in Figure 11: payment Txid_i from Alice to Edward and payment Txid_j from Alice to Fabi. The payment channel between Alice and Bob is a contending bottleneck for both Txid_i

Open channel: On input $(\mathsf{open}, c_{\langle u,u'\rangle}, v, u', t, f)$ from a user u, $\mathcal{F}$ checks whether $c_{\langle u,u'\rangle}$ is well-formed (contains valid identifiers and it is not a duplicate) and eventually sends $(c_{\langle u,u'\rangle}, v, t, f)$ to u', who can either abort or authorize the operation. In the latter case, $\mathcal{F}$ appends the tuple $(c_{\langle u,u'\rangle}, v, t, f)$ to B and the tuple $(c_{\langle u,u'\rangle}, v, t, h)$ to $\mathcal{L}$, for some random h. $\mathcal{F}$ returns h to u and u'.

Close channel: On input $(\mathsf{close}, c_{\langle u,u'\rangle}, h)$ from a user $\in \{u', u\}$ the ideal functionality $\mathcal{F}$ parses B for an entry $(c_{\langle u,u'\rangle}, v, t, f)$ and $\mathcal{L}$ for an entry $(c_{\langle u,u'\rangle}, v', t', h)$, for $h \neq \perp$. If $c_{\langle u,u'\rangle} \in \mathcal{C}$ or or $t > |\mathsf{B}|$ or $t' > |\mathsf{B}|$, the functionality aborts. Otherwise, $\mathcal{F}$ adds the entry $(c_{\langle u,u'\rangle}, v', t', f)$ to B and adds $c_{\langle u,u'\rangle}$ to $\mathcal{C}$. $\mathcal{F}$ then notifies both users involved with a message $(c_{\langle u,u'\rangle}, \perp, h)$.

Payment: On input $(\mathsf{pay}, v, (c_{\langle u_0,u_1\rangle}, \ldots, c_{\langle u_n,u_{n+1}\rangle}), (t_0, \ldots, t_n), \mathsf{Txid})$ from a user u_0, $\mathcal{F}$ executes the following interactive protocol:

1. For all $i \in \{1, \ldots, (n+1)\}$, $\mathcal{F}$ parses B for an entry of the form $((c_{\langle u_{i-1},u'_i\rangle}, v_i, t'_i, f_i))$. If such an entry does exist, $\mathcal{F}$ sends the tuple $(\mathsf{Txid}, \mathsf{Txid}, c_{\langle u_{i-1},u_i\rangle}, c_{\langle u_i,u_{i+1}\rangle}, v - \sum_{j=i}^{n} f_j, t_{i-1}, t_i)$ to the user u_i via an anonymous channel (for the specific case of the receiver the tuple is only $(\mathsf{Txid}, c_{\langle u_n,u_{n+1}\rangle}, v, t_n)$). Then, $\mathcal{F}$ checks whether for all entries of the form $(c_{\langle u_{i-1},u_i\rangle}, v'_i, \cdot, \cdot) \in \mathcal{L}$ it holds that $v'_i \geq \left(v - \sum_{j=i}^{n} f_j\right)$ and that $t_{i-1} \geq t_i$. If this is the case, $\mathcal{F}$ adds $d_i = (c_{\langle u_{i-1},u_i\rangle}, v'_i - (v - \sum_{j=i}^{n} f_j), t_i, \perp)$ to $\mathcal{L}$, where $(c_{\langle u_{i-1},u_i\rangle}, v'_i, \cdot, \cdot) \in \mathcal{L}$ is the entry with the lowest v'_i and sets $\mathsf{queued} = n + 1$. Otherwise, $\mathcal{F}$ performs the following steps:

   - If there exists an entry of the form $(c_{\langle u_k,u_{k+1}\rangle}, -, -, \mathsf{Txid}^*) \in \mathcal{L}$ such that $\mathsf{Txid} > \mathsf{Txid}^*$, then $\mathcal{F}$ adds $d_l = (c_{\langle u_{l-1},u_l\rangle}, v'_l - (v + \sum_{j=l}^{n} f_j), t_l, \perp)$ to $\mathcal{L}$, for $l \in \{1, \ldots, k\}$. Additionally, $\mathcal{F}$ adds $(\mathsf{Txid}, (c_{\langle u_k,u_{k+1}\rangle}, \ldots, c_{\langle u_n,u_{n+1}\rangle}), v - \sum_{j=k}^{n} f_j, (t_k, \ldots, t_n)) \in \mathcal{W}$. Finally, $\mathcal{F}$ sets $\mathsf{queued} = k$.

   - Otherwise, $\mathcal{F}$ removes from $\mathcal{L}$ all the entries d_i added in this phase. Additionally, $\mathcal{F}$ looks for entries of the form $(\mathsf{Txid}', (c_{\langle i,i+1\rangle}, \ldots, c_{\langle \tilde{n},\tilde{n}+1\rangle}), \tilde{v}, (t_i, \ldots, \tilde{t_n})) \in \mathcal{W}$, deletes them and execute $(\mathsf{pay}, \tilde{v}, (c_{\langle i,i+1\rangle}, \ldots, c_{\langle \tilde{n},\tilde{n}+1\rangle}), (t_i, \ldots, \tilde{t_n}))$.

2. For all $i \in \{\mathsf{queued}, \ldots, 1\}$ $\mathcal{F}$ queries all u_i with (h_i, h_{i+1}), through an anonymous channel. Each user can reply with either $\top$ or $\perp$. Let j be the index of the user that returns $\perp$ such that for all $i > j$: u_i returned $\top$. If no user returned $\perp$ we set $j = 0$.

3. For all $i \in \{j + 1, \ldots, \mathsf{queued}\}$ the ideal functionality $\mathcal{F}$ updates $d_i \in \mathcal{L}$ (defined as above) to $(-, -, -, \mathsf{Txid})$ and notifies the user of the success of the operation with with some distinguished message $(\mathsf{success}, \mathsf{Txid}, \mathsf{Txid})$. For all $i \in \{0, \ldots, j\}$ (if $j \neq 0$) $\mathcal{F}$ performs the following steps:

   - Removes d_i from $\mathcal{L}$ and notifies the user with the message $(\perp, \mathsf{Txid}, \mathsf{Txid})$.
   - $\mathcal{F}$ looks for entries of the form $(\mathsf{Txid}', (c_{\langle i,i+1\rangle}, \ldots, c_{\langle \tilde{n},\tilde{n}+1\rangle}), \tilde{v}, (t_i, \ldots, \tilde{t_n})) \in \mathcal{W}$, removes them from $\mathcal{W}$ and execute $(\mathsf{pay}, \tilde{v}, (c_{\langle i,i+1\rangle}, \ldots, c_{\langle \tilde{n},\tilde{n}+1\rangle}), (t_i, \ldots, \tilde{t_n}))$.

Figure 10: Ideal world functionality for PCNs for non-blocking progress.

and Txid_j, however, only one of the payments can be successfully executed since the payment channel between Alice and Bob has the capacity for only one of the two to be successful. Suppose by contradiction that both Txid_i and Txid_j are successfully completed. Indeed, this is possible since byzantine users Alice and Bob can respond with an incorrect payment channel capacity to users Edward and Fabi. However, the payment channel between Alice and Bob does not have sufficient capacity for both transactions to be successful—contradiction since there does not exist any equivalence to the sequential specification of payments channels. □

Proof for Lemma 2. Suppose by contradiction that there exists a strictly serializable disjoint-access implementation providing non-blocking progress. Consider the following payment network: $u_1 \to u_2 \to u_3 \to u_4 \to u_5 \to u_1$. Consider two concurrent pay operations of the form $\mathsf{pay}_1(c_{\langle u_1,u_2\rangle}, c_{\langle u_2,u_3\rangle}, c_{\langle u_3,u_4\rangle}, c_{\langle u_4,u_5\rangle}, v)$ and $\mathsf{pay}_2(c_{\langle u_4,u_5\rangle}, c_{\langle u_5,u_1\rangle}, c_{\langle u_1,u_2\rangle}, c_{\langle u_2,u_3\rangle}, v)$. Consider the execution E in which pay_1 and pay_2 run concurrently up to the following step: pay_1 executes from $u_1 \to \ldots u_4$ and pay_2 executes from $u_4 \to u_5 \to u_1$. Let E_1 (and resp. E_2) be the extensions of E in which pay_1 (and resp. pay_2) terminates *successfully* and pay_2 (and resp. pay_1) terminates *unsuccessfully*. By assumption of non-blocking progress, there exists such a finite extension of this execution in which both pay_1 and pay_2 must terminate (though they may not be successful since this depends on the available channel capacity).

Since the implementation is disjoint-access parallel, execution E_1 is *indistinguishable* to $(u_1, \ldots, u_5)$ (and resp. $(u_4, \ldots, u_3)$) from the execution $\bar{E}$, an extension of E, in which only pay_1 (and resp. pay_2) is successful *and* matches the sequential specification of PCN. Note that analogous arguments applies for the case of E_2.

However, E_1 (and resp. E_2) is not a correct execution since it lacks the *all-or-nothing* semantics: only a proper subset of the channels from the execution E involved in pay_1 (and resp. pay_2) have their capacities decreased by v (and resp. v'). This is a contradiction to the assumption of strict serializability, thus completing the proof. □